BỘ GIAO THÔNG VẬN TẢI
TRƯỜNG ĐẠI HỌC HÀNG HẢI
BỘ MÔN: KHOA HỌC MÁY TÍNH
KHOA: CÔNG NGHỆ THÔNG TIN

BÀI GIẢNG
PHÂN TÍCH THIẾT KẾ VÀ ĐÁNH GIÁ THUẬT TOÁN

TÊN HỌC PHẦN : **Phân tích thiết kế và đánh giá thuật toán**
MÃ HỌC PHẦN : 17208
TRÌNH ĐỘ ĐÀO TẠO : **ĐẠI HỌC CHÍNH QUY**
DÙNG CHO SV NGÀNH : **CÔNG NGHỆ THÔNG TIN**

HẢI PHÒNG - 2010

Tên học phần: Phân tích thiết kế và đánh giá thuật toán **Loại học phần:** 2
Bộ môn phụ trách giảng dạy: Khoa học Máy tính **Khoa phụ trách:** CNTT
Mã học phần: 17208 **Tổng số TC:** 3

TS tiết	Lý thuyết	Thực hành/Xemina	Tự học	Bài tập lớn	Đồ án môn học
60	45	15	0	0	0

Điều kiện tiên quyết:

Sinh viên phải học xong các học phần sau mới được đăng ký học phần này:
Kỹ thuật lập trình, Cấu trúc dữ liệu, Toán rời rạc

Mục tiêu của học phần:

- Cung cấp các kiến thức cơ bản về thuật toán, cấu trúc dữ liệu.
- Cung cấp các kiến thức về chiến lược xây dựng và đánh giá thuật toán
- Rèn luyện tư duy khoa học.

Nội dung chủ yếu

Gồm 4 phần:

- Các kiến thức cơ bản về thuật toán.
- Các kiến thức cơ bản về sắp xếp và tìm kiếm dữ liệu.
- Các chiến lược thiết kế thuật toán: chiến lược chia để trị, chiến lược quay lui, chiến lược qui hoạch động, chiến lược tham lam
- Kiến thức cơ bản về đánh giá độ phức tạp thuật toán.

Nội dung chi tiết của học phần:

TÊN CHƯƠNG MỤC	PHÂN PHỐI SỐ TIẾT				
	TS	**LT**	**TH/Xemina**	**BT**	**KT**
Chương I. Các khái niệm cơ bản	5	4	0	1	0
1.1. Giới thiệu về thuật toán 1.1.1 Khái niệm về thuật toán. 1.1.2. Các phương pháp biểu diễn thuật toán 1.1.3. Các ví dụ biểu diễn thuật toán sơ đồ khối		1			
1.2. Độ phức tạp thuật toán 1.2.1. Các ký hiệu, hàm đánh giá độ phức tạp thuật toán		2		1	
1.2.2. Các lớp thuật toán		0,5			
1.3. Mối quan hệ giữa cấu trúc dữ liệu và giải thuật 1.4. Một số ví dụ		0,5			
Chương II. Sắp xếp và tìm kiếm	15	7	5	2	1
2.1. Bài toán sắp xếp 2.1.1. Sắp xếp trong 2.1.2. Sắp xếp ngoài 2.1.3. Đánh giá thuật toán sắp xếp		0,5			
2.2. Các thuật toán sắp xếp cơ bản 2.2.1. Sắp xếp chọn (Selection Sort) 2.2.2. Sắp xếp đổi chỗ trực tiếp (Exchange Sort) 2.2.3. Sắp xếp chèn (Insertion Sort) 2.2.4. Sắp xếp nổi bọt (Bubble Sort) 2.2.5. So sánh các thuật toán sắp xếp cơ bản		3	2,5	1	

TÊN CHƯƠNG MỤC	PHÂN PHỐI SỐ TIẾT				
	TS	LT	TH/Xemina	BT	KT
2.3. Sắp xếp vun đống 2.3.1. Cấu trúc Heap 2.3.2. Thuật toán xây dựng cấu trúc Heap 2.3.3. Thuật toán sắp xếp vun đống 2.4. Tìm kiếm tuyến tính 2.4.1. Bài toán tìm kiếm 2.4.2. Thuật toán tìm kiếm tuyến tính		2,5 1	1,5 1	1	
Chương III. Đệ qui và chiến lược vét cạn	11	6	3	2	0
3.1. Khái niệm về đệ quy 3.1.1. Giải thuật đệ quy và thủ tục đệ quy 3.1.2. Thiết kế giải thuật đệ quy 3.1.3. Hiệu lực của đệ quy. 3.1.4. Đệ quy và quy nạp toán học. 3.2. Chiến lược vét cạn (Bruteforce) 3.3. Chiến lược đệ qui quay lui (backtracking) 3.3.1. Vector nghiệm 3.3.2. Thủ tục đệ qui 3.3.3. Các giá trị đề cử 3.3.4. Điều kiện chấp nhận 3.3.5. Một số bài toán backtracking điển hình		1 0,5 1 1 1 0,5 1	 1 2	1 1	
Chương IV. Chiến lược chia để trị	11	6	3	1	1
4.1. Cơ sở của chiến lược chia để trị 4.2. Thuật toán sắp xếp bằng trộn 4.2.1. Thuật toán trộn hai Run 4.2.2. Sắp xếp bằng trộn 4.3. Sắp xếp nhanh (Quick sort) 4.3.1. Chiến lược phân hoạch 4.3.2. Quick sort 4.4. Tìm kiếm nhị phân 4.5. Thuật toán nhân số nguyên 4.5.1. Thuật toán nhân tay 4.5.2. Thuật toán chia để trị 4.6. Một số bài toán khác		0,5 1,5 1,5 1 1 0,5	 1 1 1	 1	
Chương V. Qui hoạch động	12	6	3	2	1
5.1. Chiến lược qui hoạch động 5.1.1. Các điều kiện để áp dụng 5.1.2. Các bước trong qui hoạch động 5.1.3. Các kiểu qui hoạch động 5.2. Bài toán dãy số Fibonaci 5.2.1. Thuật toán đệ qui 5.2.2. Thuật toán qui hoạch động 5.3. Bài toán dãy con chung dài nhất 5.4. Bài toán nhân ma trận 5.5. Một số ví dụ khác		1,5 1 1 1,5 1	 0,5 1 1 0,5	 1 1	
Chương VI. Chiến lược tham lam	6	4	1	1	0
6.1. Nguyên tắc tham lam 6.2. Bài toán đổi tiền 6.3. Bài toán sắp lịch các sự kiện 6.3.1. Thuật toán đệ qui		0,5 1 2	 1		

TÊN CHƯƠNG MỤC	PHÂN PHỐI SỐ TIẾT				
	TS	LT	TH/Xemina	BT	KT
6.3.2. Thuật toán theo chiến lược tham lam 6.4. So sánh chiến lược tham lam với chiến lược qui hoạch động		0,5		1	

Nhiệm vụ của sinh viên :

Tham dự các buổi thuyết trình của giáo viên, tự học, tự làm bài tập do giáo viên giao, tham dự các bài kiểm tra định kỳ và cuối kỳ.

Tài liệu học tập :

- Nguyễn Hữu Điển, *Giáo trình một số vấn đề về thuật toán*, NXB Giáo dục, 2003
- Đinh Mạnh Tường. *Cấu trúc dữ liệu và thuật toán*. NXB Đại học Quốc gia Hà nội. 2002.
- Nguyễn Quốc Lượng, Hoàng Đức Hải. *Cấu trúc dữ liệu + giải thuật = chương trình*. NXB Giáo dục. 1996
- Richard Neapolitan và Kumarss Naimipour, *Foundations of Algorithms Using C++ Pseudocode, Third Edition*, Jones and Bartlett Publishers, 2004.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms, Second Edition*, MIT Press, 2001.

Hình thức và tiêu chuẩn đánh giá sinh viên:

- Hình thức thi cuối kỳ : Thi vấn đáp.
- Sinh viên phải đảm bảo các điều kiện theo Quy chế của Nhà trường và của Bộ

Thang điểm: Thang điểm chữ A, B, C, D, F

Điểm đánh giá học phần: Z = 0,3X + 0,7Y.

Bài giảng này là tài liệu **chính thức và thống nhất** của Bộ môn Khoa học Máy tính, Khoa Công nghệ Thông tin và được dùng để giảng dạy cho sinh viên.

Ngày phê duyệt: / /20

Trưởng Bộ môn: ThS. Nguyễn Hữu Tuân (ký và ghi rõ họ tên)

MỤC LỤC

LỜI NÓI ĐẦU 1
CHƯƠNG I: CÁC KHÁI NIỆM CƠ BẢN 2
1. Thuật toán (giải thuật) - Algorithm 2
1.1. Định nghĩa thuật toán 2
1.2. Đặc trưng của thuật toán 2
2. Biểu diễn thuật toán 2
2.1. Mô tả các bước thực hiện 2
2.2. Sử dụng sơ đồ (lưu đồ) giải thuật (flowchart) 3
3. Độ phức tạp thuật toán – Algorithm Complexity 4
3.1. Các tiêu chí đánh giá thuật toán 4
3.2. Đánh giá thời gian thực hiện thuật toán 4
3.3. Các định nghĩa hình thức về độ phức tạp thuật toán 5
3.4. Các lớp thuật toán 7
4. Cấu trúc dữ liệu – Data structure 9
5. Các chiến lược thiết kế thuật toán. 9
5.1. Duyệt toàn bộ (Exhausted search) 9
5.2. Đệ qui quay lui – Backtracking 9
5.3. Chia để trị (Divide and Conquer) 9
5.4. Chiến lược tham lam (Greedy) 10
5.5. Qui hoạch động (Dynamic Programming) 11
6. Bài tập 11
CHƯƠNG II: SẮP XẾP (SORTING) VÀ TÌM KIẾM (SEARCHING) 13
1. Bài toán sắp xếp 13
1.1. Sắp xếp trong (Internal Sorting) 13
1.2. Sắp xếp ngoài (External Sorting) 13
1.3. Sắp xếp gián tiếp 13
1.3. Các tiêu chuẩn đánh giá một thuật toán sắp xếp 14
2. Các phương pháp sắp xếp cơ bản 15
2.1. Sắp xếp chọn (Selection sort) 15
2.2. Sắp xếp đổi chỗ trực tiếp (Exchange sort) 17
2.3. Sắp xếp chèn (Insertion sort) 19
2.4. Sắp xếp nổi bọt (Bubble sort) 21

2.5. So sánh các thuật toán sắp xếp cơ bản 23
3. Cấu trúc dữ liệu Heap, sắp xếp vun đống (Heap sort). 24
4. Tìm kiếm tuyến tính 31
5. Các vấn đề khác 33
6. Bài tập 33
CHƯƠNG III: ĐỆ QUI VÀ CHIẾN LƯỢC VÉT CẠN 34
1. Khái niệm đệ qui 34
2. Chiến lược vét cạn (Brute force) 34
3. Chiến lược quay lui (Back tracking / try and error) 35
CHƯƠNG IV: CHIẾN LƯỢC CHIA ĐỂ TRỊ 38
1. Cơ sở của chiến lược chia để trị (Divide and Conquer) 38
2. Sắp xếp trộn (Merge sort) 38
3. Sắp xếp nhanh (Quick sort) 43
4. Tìm kiếm nhị phân 46
5. Bài tập 48
CHƯƠNG V: QUI HOẠCH ĐỘNG 49
1. Chiến lược qui hoạch động 49
2. Bài toán 1: Dãy Fibonaci 49
3. Bài toán 2: Bài toán nhân dãy các ma trận 51
4. Phương pháp qui hoạch động 53
5. Bài toán dãy con chung dài nhất 53
6. Bài tập 57
CHƯƠNG VI: CHIẾN LƯỢC THAM LAM (GREEDY) 60
1. Nguyên tắc tham lam 60
2. Bài toán đổi tiền 60
3. Bài toán lập lịch 61
4. So sánh chiến lược tham lam và qui hoạch động 64
TÀI LIỆU THAM KHẢO 65
ĐỀ THI THAM KHẢO 66

LỜI NÓI ĐẦU

Cấu trúc dữ liệu và các chiến lược thiết kế thuật toán là các lĩnh vực nghiên cứu gắn liền với nhau và là một trong những lĩnh vực nghiên cứu lâu đời của khoa học máy tính. Hầu hết các chương trình được viết ra, chạy trên máy tính, dù lớn hay nhỏ, dù đơn giản hay phức tạp, đều phải sử dụng các cấu trúc dữ liệu tuân theo các trình tự, cách thức làm việc nào đó, chính là các giải thuật. Việc hiểu biết về các thuật toán và các chiên lược xây dựng thuật toán cho phép các lập trình viên, các nhà khoa học máy tính có nền tảng lý thuyết vững chắc, có nhiều lựa chọn hơn trong việc đưa ra các giải pháp cho các bài toán thực tế. Vì vậy việc học tập môn học Phân tích thiết kế và dánh giá giải thuật là một điều quan trọng.

Tài liệu này dựa trên những kinh nghiệm và nghiên cứu mà tác giả đã đúc rút, thu thập trong quá trình giảng dạy môn học Cấu trúc dữ liệu và giải thuật tại khoa Công nghệ Thông tin, Đại học Hàng hải Việt nam, cùng với sự tham khảo của các tài liệu của các đồng nghiệp, các tác giả trong và ngoài nước, từ điển trực tuyến Wikipedia. Với bẩy chương được chia thành các chủ đề khác nhau từ các khái niệm cơ bản cho tới thuật toán sắp xếp, tìm kiếm, các chiến lược thiết kế thuật toán như đệ qui, quay lui, qui hoạch động, tham lam … hy vọng sẽ cung cấp cho các em sinh viên, các bạn độc giả một tài liệu bổ ích. Mặc dù đã rất cố gắng song vẫn không tránh khỏi một số thiếu sót, hy vọng sẽ được các bạn bè đồng nghiệp, các em sinh viên, các bạn độc giả góp ý chân thành để tôi có thể hoàn thiện hơn nữa tài liệu này.

Xin gửi lời cảm ơn chân thành tới các bạn bè đồng nghiệp và Ban chủ nhiệm khoa Công nghệ Thông tin đã tạo điều kiện giúp đỡ để tài liệu này có thể hoàn thành.

Hải phòng, tháng 04 năm 2010

Tác giả

Nguyễn Hữu Tuân

CHƯƠNG I: CÁC KHÁI NIỆM CƠ BẢN

1. Thuật toán (giải thuật) - Algorithm

1.1. Định nghĩa thuật toán

Có rất nhiều các định nghĩa cũng như cách phát biểu khác nhau về định nghĩa của thuật toán. Theo như cuốn sách giáo khoa nổi tiếng viết về thuật toán là "**Introduction to Algorithms**" (Second Edition của Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest và Clifford Stein) thì thuật toán được định nghĩa như sau: "một thuật toán là một thủ tục tính toán xác định (well-defined) nhận các giá trị hoặc một tập các giá trị gọi là input và sinh ra ra một vài giá trị hoặc một tập giá trị được gọi là output".

Nói một cách khác các thuật toán giống như là các cách thức, qui trình để hoàn thành một công việc cụ thể xác định (well-defined) nào đó. Vì thế một đoạn mã chương trình tính các phần tử của dãy số Fibonaci là một cài đặt của một thuật toán cụ thể. Thậm chí một hàm đơn giản để cộng hai số cũng là một thuật toán hoàn chỉnh, mặc dù đó là một thuật toán đơn giản.

1.2. Đặc trưng của thuật toán

Tính đúng đắn: Thuật toán cần phải đảm bảo cho một kết quả đúng sau khi thực hiện đối với các bộ dữ liệu đầu vào. Đây có thể nói là đặc trưng quan trọng nhất đối với một thuật toán.

Tính dừng: thuật toán cần phải đảm bảo sẽ dừng sau một số hữu hạn bước.

Tính xác định: Các bước của thuật toán phải được phát biểu rõ ràng, cụ thể, tránh gây nhập nhằng hoặc nhầm lẫn đối với người đọc và hiểu, cài đặt thuật toán.

Tính hiệu quả: thuật toán được xem là hiệu quả nếu như nó có khả năng giải quyết hiệu quả bài toán đặt ra trong thời gian hoặc các điều kiện cho phép trên thực tế đáp ứng được yêu cầu của người dùng.

Tính phổ quát: thuật toán được gọi là có tính phố quát (phổ biến) nếu nó có thể giải quyết được một lớp các bài toán tương tự.

Ngoài ra mỗi thuật toán theo định nghĩa đều nhận các giá trị đầu vào được gọi chung là các giá trị dữ liệu Input. Kết quả của thuật toán (thường là một kết quả cụ thể nào đó tùy theo các bài toán và thuật toán cụ thể) được gọi là Output.

2. Biểu diễn thuật toán

Thường có hai cách biểu diễn một thuật toán, cách thứ nhất là mô tả các bước thực hiện của thuật toán, cách thứ hai là sử dụng sơ đồ giải thuật.

2.1. Mô tả các bước thực hiện

Để biểu diễn thuật toán người ta mô tả chính xác các bước thực hiện của thuật toán, ngôn ngữ dùng để mô tả thuật toán có thể là ngôn ngữ tự nhiên hoặc một ngôn ngữ lai ghép giữa ngôn ngữ tự nhiên với một ngôn ngữ lập trình nào đó gọi là các đoạn giả mã lệnh.

Ví dụ: mô tả thuật toán tìm ước số chung lớn nhất của hai số nguyên.

Input: Hai số nguyên a, b.

Output: Ước số chung lớn nhất của a, b.

Thuật toán:

Bước 1: Nếu a=b thì USCLN(a, b)=a.

Bước 2: Nếu a > b thì tìm USCLN của a-b và b, quay lại bước 1;

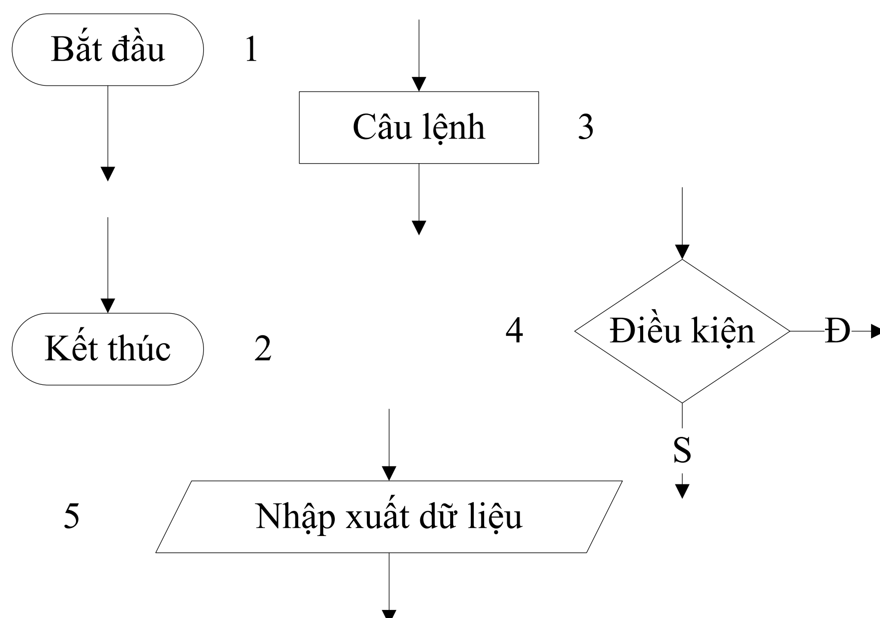
Bước 3: Nếu a < b thì tìm USCLN của a và b-a, quay lại bước 1;

2.2. Sử dụng sơ đồ (lưu đồ) giải thuật (flowchart)

Một trong những cách phổ biến để biểu diễn thuật toán là sử dụng sơ đồ thuật toán (Algorithm Flowchart).

Sơ đồ thuật toán sử dụng các ký hiệu hình khối cơ bản để tạo thành một mô tả mang tính hình thức (cách này rõ ràng hơn so với việc mô tả các bước thực hiện thuật toán) của thuật toán. Chúng ta có thể hình dung việc sử dụng sơ đồ giải thuật để mô tả thuật toán giống như dùng các bản vẽ để mô tả cấu trúc của các tòa nhà.

Các khối cơ bản của một sơ đồ thuật toán



Khối 1: Khối bắt đầu thuật toán, chỉ có duy nhất một đường ra;

Khối 2: Khối kết thúc thuật toán, có thể có nhiều đường vào;

Khối 3: Thực hiện câu lệnh (có thể là một hoặc nhiều câu lệnh); gồm một đường vào và một đường ra;

Khối 4: Rẽ nhánh, kiểm tra biểu thức điều kiện (biểu thức Boolean), nếu biểu thức đúng thuật toán sẽ đi theo nhánh Đúng (True), nếu biểu thức sai thuật toán sẽ đi theo nhánh Sai (False).

Khối 5: Các câu lệnh nhập và xuất dữ liệu.

3. Độ phức tạp thuật toán – Algorithm Complexity

3.1. Các tiêu chí đánh giá thuật toán

Thông thường để đánh giá mức độ tốt, xấu và so sánh các thuật toán cùng loại, có thể dựa trên hai tiêu chuẩn:

+ Thuật toán đơn giản, dễ hiểu, dễ cài đặt.

+ Dựa vào thời gian thực hiện và tài nguyên mà thuật toán sử dụng để thực hiện trên các bộ dữ liệu.

Trên thực tế các thuật toán hiệu quả thì không dễ hiểu, các cài đặt hiệu quả cũng không dễ dàng thực hiện và hiểu được một cách nhanh chóng. Và một điều có vẻ nghịch lý là các thuật toán càng hiệu quả thì càng khó hiểu, cài đặt càng phức tạp lại càng hiệu quả (không phải lúc nào cũng đúng). Vì thế để đánh giá và so sánh các thuật toán người ta thường dựa trên độ phức tạp về thời gian thực hiện của thuật toán, gọi là độ phức tạp thuật toán (**algorithm complexity**). Về bản chất độ phức tạp thuật toán là một hàm ước lượng (có thể không chính xác) số phép tính mà thuật toán cần thực hiện (từ đó dễ dàng suy ra thời gian thực hiện của thuật toán) đối với một bộ dữ liệu input có kích thước N. N có thể là số phần tử của mảng trong trường hợp bài toán sắp xếp hoặc tìm kiếm, hoặc có thể là độ lớn của số trong bài toán kiểm tra số nguyên tố chẳng hạn.

3.2. Đánh giá thời gian thực hiện thuật toán

Để minh họa việc đánh giá độ phức tạp thuật toán ta xem xét ví dụ về thuật toán sắp xếp chọn (selection sort) và sắp xếp đổi chỗ trực tiếp (exchange sort) như sau:

Cài đặt của thuật toán sắp xếp chọn:

```
for(i=0;i<n;i++)
{
  min_idx = i;
  for(j=i+1;j<n;j++)
          if(a[j]<a[min_idx])
                  min_idx = j;
  if(min_idx!=i)
        {
          temp = a[i];
          a[i] = a[min_idx];
          a[min_idx] = temp;
        }
```

}

Số phép tính thuật toán cần thực hiện được tính như sau:

(N-1) + (N-2) + … + 2 + 1 = N*(N-1)/2.

Phân tích chi tiết hơn thì N*(N-1)/2 là số phép toán so sánh cần thực hiện, còn số lần thực hiện đổi chỗ hai phần tử (số nguyên) tối đa của thuật toán là N.

Cài đặt của thuật toán sắp xếp đổi chỗ trực tiếp:

```
for(i=0;i<n;i++)
        for(j=i+1;j<n;j++)
  if(a[j] < a[i])
        {
          temp = a[i];
          a[i] = a[j];
          a[j] = temp;
        }
```

Tương tự đối với thuật toán sắp xếp chọn ta có số phép toán thực hiện là: (N-1) + (N-2) + …+2 +1 = N*(N-1)/2. Chi tiết hơn, N*(N-1)/2 là số lần so sánh thuật toán thực hiện, và cũng là số lần đổi chỗ hai phần tử (hai số nguyên) tối đa của thuật toán.

Trong trường hợp trung bình, thuật toán sắp xếp chọn có xu hướng tốt hơn so với sắp xếp đổi chỗ trực tiếp vì số thao tác đổi chỗ ít hơn, còn trong trường hợp tốt nhất thì như nhau, trường hợp tồi nhất thì chắc chắn thuật toán sắp xếp chọn tốt hơn, do đó có thể kết luận thuật toán sắp xếp chọn nhanh hơn so với thuật toán sắp xếp đổi chỗ trực tiếp.

3.3. Các định nghĩa hình thức về độ phức tạp thuật toán

Gọi f, g là các hàm không giảm định nghĩa trên tập các số nguyên dương. (chú ý là tất cả các hàm thời gian đều thỏa mãn các điều kiện này). Chúng ta nói rằng hàm f(N) là $\boldsymbol{O}$(g(N)) (đọc là: f là O lớn của g) nếu như tồn tại một hằng số c và N_0:

$$\forall N > N_0; f(N) < c.g(N)$$

Phát biểu thành lời như sau: f(N) là $\boldsymbol{O}$(g(N)) nếu tồn tại c sao cho hầu hết phần đồ thị của hàm f nằm dưới phần đồ thị của hàm c*g. Chú ý là hàm f tăng nhiều nhất là nhanh bằng hàm c*g.

Thay vì nói f(N) là $\boldsymbol{O}$(g(N)) chúng ta thường viết là f(N) = $\boldsymbol{O}$(g(N)). Chú ý rằng đẳng thức này không có tính đối xứng có nghĩa là chúng ta có thể viết ngược lại $\boldsymbol{O}$(g(N)) = f(N) nhưng không thể suy ra g(N) = $\boldsymbol{O}$(f(N)).

Định nghĩa trên được gọi là ký hiệu O lớn (big-O notation) và thường được sử dụng để chỉ định các chặn trên của hàm tăng.

Chẳng hạn đối với ví dụ về sắp xếp bằng chọn ta có $f(N) = N*(N-1)/2 = 0.5N^2 - 0.5N$ chúng ta có thể viết là $f(N) = \boldsymbol{O}(N^2)$. Có nghĩa là hàm f không tăng nhanh hơn hàm N^2.

Chú ý rằng thậm chí hàm f chính xác có công thức như thế nào không cho chúng ta câu trả lời chính xác của câu hỏi "Chương trình có thời gian thực hiện là bao lâu trên máy tính của tôi?". Nhưng điều quan trọng là qua đó chúng ta biết được hàm thời gian thực hiện của thuật toán là hàm bậc hai. Nếu chúng ta tăng kích thước input lên 2 lần, thời gian thực hiện của chương trình sẽ tăng lên xấp xỉ 4 lần không phụ thuộc vào tốc độ của máy.

Chặn trên $f(N) = \boldsymbol{O}(N^2)$ cho chúng ta kết quả gần như thế – nó đảm bảo rằng độ tăng của hàm thời gian nhiều nhất là bậc hai.

Do đó chúng ta sẽ sử dụng ký pháp O lớn để mô tả thời gian thực hiện của thuật toán (và đôi khi cả bộ nhớ mà thuật toán sử dụng). Đối với thuật toán trong ví dụ 2 chúng ta có thể nói "độ phức tạp thời gian của thuật toán là $\boldsymbol{O}(N^2)$ hoặc ngắn gọn là "thuật toán là $\boldsymbol{O}(N^2)$".

Tương tự chúng ta có các định nghĩa Ω (**omega**)và Θ (**theta**):

Chúng ta nói rằng hàm f(N) là Ω(g(N)) nếu như $g(N) = \boldsymbol{O}(f(N))$, hay nói cách khác hàm f tăng ít nhất là nhanh bằng hàm g.

Và nói rằng hàm f(N) là Θ(g(N)) nếu như $f(N) = \boldsymbol{O}(g(N))$ và $g(N) = \boldsymbol{O}(f(N))$, hay nói cách khác cả hai hàm xấp xỉ như nhau về độ tăng.

Hiển nhiên là cách viết Ω là để chỉ ra chặn dưới và Θ là để chỉ ra một giới hạn chặt chẽ của một hàm. Còn có nhiều giới hạn khác nữa nhưng đây là các giới hạn mà chúng ta hay gặp nhất.

Một vài ví dụ:

- $0.5N^2 - 0.5N = \boldsymbol{O}(N2)$
- $47\ N*\log(N) = \boldsymbol{O}(N*\log(N))$
- $N*\log(N) + 1000047N = \Theta(N*\log(N))$
- Tất cả các hàm đa thức bậc k đều là $\boldsymbol{O}(N^k)$
- Độ phức tạp thời gian của thuật toán sắp xếp chọn và sắp xếp đổi chỗ trực tiếp là $\Theta(N^2)$
- Nếu một thuật toán là $\boldsymbol{O}(N^2)$ thì nó cũng là $\boldsymbol{O}(N^5)$
- Mỗi thuật toán sắp xếp dựa trên so sánh có độ phức tạp tối ưu là $\Omega(N*\log(N))$
- Thuật toán sắp xếp MergeSort có số thao tác so sánh là N*log(N). Do đó độ phức tạp thời gian của MergeSort là Θ(N*log(N)) có nghĩa là MergeSort là tiệm cận với thuật toán sắp xếp tối ưu.

Khi xem xét so sánh các thuật toán cùng loại người ta thường xét độ phức tạp của thuật toán trong các trường hợp: trung bình (average case), trường hợp xấu nhất (the worst case) và trường hợp tốt nhất (the best case).

3.4. Các lớp thuật toán

Khi chúng ta nói về độ phức tạp thời gian/ không gian nhớ của một thuật toán thay vì sử dụng các ký hiệu hình thức $\Theta(f(n))$ chúng ta có thể đơn giản đề cập tới lớp của hàm f. Ví dụ $f(N) = \Theta(N)$ chúng ta sẽ nói thuật toán là tuyến tính (linear). Có thể tham khảo thêm:

$f(N) = 1$: hằng số (constant)

$f(N) = \Theta(\log(N))$: logarit

$f(N) = \Theta(N)$: tuyến tính (linear)

$f(N) = \Theta(N*\log(N))$: N log N

$f(N) = \Theta(N^2)$: bậc hai (quadratic)

$f(N) = \Theta(N^3)$: bậc 3 (cubic)

$f(N) = \boldsymbol{O}(N^k)$ với k là một số nguyên dương: đa thức (polynomial)

$f(N) = \Omega(b^N)$: hàm mũ (exponential)

Đối với các bài toán đồ thị độ phức tạp $\Theta(V+E)$ có nghĩa là "tuyến tính đối với kích thước của đồ thị".

Xác định thời gian thực hiện từ một giới hạn tiệm cận

Đối với hầu hết các thuật toán chúng ta có thể gặp các hằng số bị che đi bởi cách viết ***O*** hoặc **b** thường là khá nhỏ. Chẳng hạn nếu độ phức tạp thuật toán là $\Theta(N^2)$ thì chúng ta sẽ mong muốn chính xác độ phức tạp thời gian là $10N^2$ chứ không phải là 10^7N^2.

Có nghĩa là nếu hằng số là lớn thì thường là theo một cách nào đó liên quan tới một vài hằng số của bài toán. Trong trường hợp này tốt nhất là gán cho hằng đó một cái tên và đưa nó vào ký hiệu tiệm cận của hằng số đó.

Ví dụ: bài toán đếm số lần xuất hiện của mỗi ký tự trong một xâu có N ký tự. Một thuật toán cơ bản là duyệt qua toàn bộ xâu đối với mỗi ký tự để thực hiện đếm xem ký tự đó xuất hiện bao nhiêu lần. Kích thước của bảng chữ cái là cố định (nhiều nhất là 255 đối với ngôn ngữ lập trình C) do đó thuật toán là tuyến tính đối với N. Nhưng sẽ là tốt hơn nếu viết là độ phức tạp của thuật toán là $\Theta(S*N)$ trong đó S là số phần tử của bảng chữ cái sử dụng. (Chú ý là có một thuật toán tốt hơn để giải bài toán này với độ phức tạp là $\Theta(S + N)$.

Trong các cuộc thi lập trình một thuật toán thực hiện 1000000000 phép nhân có thể không thỏa mãn ràng buộc thời gian. Chúng ta có thể tham khảo bảng sau để biết thêm:

Độ phức tạp	Giá trị N lớn nhất
Θ (N)	100 000 000
Θ (N log N)	40 000 000
Θ (N2)	10 000
Θ (N3)	500

Θ (N4)	90
Θ (2N)	20
Θ (N!)	11

Thời gian thực hiện của các thuật toán có độ phức tạp khác nhau	
O(Log(N))	10^{-7} giây
O(N)	10^{-6} giây
O(N*Log(N))	10^{-5} giây
O(N^2)	10^{-4} giây
O(N^6)	3 phút
O(2^N)	10^{14} năm
O(N!)	10^{142} năm

Chú ý về phân tích thuật toán.

Thông thường khi chúng ta trình bày một thuật toán cách tốt nhất để nói về độ phức tạp thời gian của nó là sử dụng các chặn Θ. Tuy nhiên trên thực tế chúng ta hay dùng ký pháp big-O – các kiểu khác không có nhiều giá trị lắm, vì cách này rất dễ gõ và cũng được nhiều người biết đến và hiểu rõ hơn. Nhưng đừng quên là big-O là chặn trên và thường thì chúng ta sẽ tìm môt chặn trên càng nhỏ càng tốt.

Ví dụ: Cho một mảng đã được sắp A. Hãy xác định xem trong mảng A có hai phần tử nào mà hiệu của chúng bằng D hay không. Hãy xem đoạn mã chương trình sau:

```
int j=0;
for (int i=0; i<N; i++)
{
          while  ( (j<N-1) && (A[i]-A[j]  > D) )
            j++;
  if (A[i]-A[j]  == D)
          return 1;
}
```

Rất dễ để nói rằng thuật toán trên là ***O***(N^2): vòng lặp while bên trong được gọi đến N lần, mỗi lần tăng j lên tối đa N lần. Nhưng một phân tích tốt hơn sẽ cho chúng ta thấy rằng thuật toán là ***O***(N) vì trong cả thời gian thực hiện của thuật toán lệnh tăng j không chạy nhiều hơn N lần.

Nếu chúng ta nói rằng thuật toán là $O(N^2)$ chúng ta vẫn đúng nhưng nếu nói là thuật toán là $O(N)$ thì chúng ta đã đưa ra được thông tin chính xác hơn về thuật toán.

4. Cấu trúc dữ liệu – Data structure

Niklaus Wirth, một lập trình viên và nhà khoa học máy tính, người phát minh ra ngôn ngữ lập trình Pascal đã từng nói một câu nói nổi tiếng trong lĩnh vực lập trình: Chương trình (Programs) = Cấu trúc dữ liệu (Data Structures) + Giải thuật (Algorithms). Câu nói này nói lên bản chất của việc lập trình là đi tìm một cấu trúc dữ liệu phù hợp để biểu diễn dữ liệu của bài toán và từ đó xây dựng giải thuật phù hợp với cấu trúc dữ liệu đã chọn. Ngày nay với sự phát triển của các kỹ thuật lập trình, câu nói của Wirth không hẳn còn đúng tuyệt đối nữa nhưng nó vẫn phản ánh sự gắn kết và tầm quan trọng của các cấu trúc dữ liệu và giải thuật. Cấu trúc dữ liệu được sử dụng để biểu diễn dữ liệu còn các giải thuật được sử dụng để thực hiện các thao tác trên các dữ liệu của bài toán nhằm hoàn thành các chức năng của chương trình

5. Các chiến lược thiết kế thuật toán.

Không có một phương pháp nào có thể giúp chúng ta xây dựng (thiết kế) nên các thuậ toán cho tất cả các loại bài toán. Các nhà khoa học máy tính đã nghiên cứu và đưa ra các chiến lược thiết kế các giải thuật chung nhất áp dụng cho các loại bài toán khác nhau.

5.1. Duyệt toàn bộ (Exhausted search)

Chiến lược duyệt toàn bộ là chiến lược mà mỗi lập trình viên phải nghĩ đến đầu tiên khi giải quyết bất cứ bài toán nào. Trong phương pháp duyệt toàn bộ, chúng ta sẽ xem xét tất cả các ứng cử viên thuộc một không gian có thể có của bài toán để xem đó có phải là nghiệm của bài toán hay không. Phương pháp này yêu cầu có một hàm kiểm tra xem một ứng cử viên nào đó có phải là nghiệm của bài toán hay không. Mặc dù dễ hiểu song phương pháp này không phải là dễ thực hiện, và đặc biệt là không hiệu quả đối với các bài toán mà kích thước input lớn. Có nhiều phương pháp cải tiến hiệu năng của phương pháp duyệt toàn bộ và chúng ta sẽ xem xét kỹ hơn trong chương 3.

5.2. Đệ qui quay lui – Backtracking

Chiến lược đệ qui quay lui là một chiến lược xây dựng thuật toán dựa trên quan hệ đệ qui. Nghiệm của bài toán được mô hình hóa dưới dạng một vecto, mỗi thành phần của vecto nghiệm sẽ có một tập giá trị có thể nhận và thuật toán sẽ tiến hành các bước gán các giá trị có thể cho các thành phần của nghiệm để xác định đúng nghiệm của bài toán. Mặc dù không phải bài toán nào cũng có thể áp dụng song các thuật giải dựa trên phương pháp đệ qui quay lui luôn có vẻ đẹp từ sự ngắn gọn, súc tích mà nó mang lại.

5.3. Chia để trị (Divide and Conquer)

Chiến lược chia để trị là một chiến lược quan trọng trong việc thiết kế các giải thuật. Ý tưởng của chiến lược này nghe rất đơn giản và dễ nhận thấy, đó là: khi cần giải quyết một bài

toán, ta sẽ tiến hành chia bài toán đó thành các bài toán nhỏ hơn, giải các bài toán nhỏ hơn đó, sau đó kết hợp nghiệm của các bài toán nhỏ hơn đó lại thành nghiệm của bài toán ban đầu.

Tuy nhiên vấn đề khó khăn ở đây nằm ở hai yếu tố: làm thế nào để chia tách bài toán một cách hợp lý thành các bài toán con, vì nếu các bài toán con lại được giải quyết bằng các thuật toán khác nhau thì sẽ rất phức tạp, yếu tố thứ hai là việc kết hợp lời giải của các bài toán con sẽ được thực hiện như thế nào?.

Các thuật toán sắp xếp trộn (merge sort), sắp xếp nhanh (quick sort) đều thuộc loại thuật toán chia để trị (các thuật toán này được trình bày ở chương 3).

Ví dụ[6, trang 57]: Trong ví dụ này chúng ta sẽ xem xét thuật toán tính a^N.

Để tính a^N ta để ý công thức sau:

N = 0

$a^N =$ Ũu N ch½n

nŨu N lÎ

Từ công thức trên ta suy ra cài đặt của thuật toán như sau:

```
int power(int a, int n)
{
  if(n==0)
          return 1;
  else{
          int t = power(a, n/2);
          if(n%2==0)
                  return t*t;
          else
                  return a*t*t;
  }
}
```

5.4. Chiến lược tham lam (Greedy)

Chiến lược tham lam là một chiến lược xây dựng thuật toán tìm nghiệm tối ưu cục bộ cho các bài toán tối ưu nhằm đạt được nghiệm tối ưu toàn cục cho cả bài toán (trong trường hợp tổng quát). Trong trường hợp cho nghiệm đúng, lời giải của chiến lược tham lam thường rất dễ cài đặt và có hiệu năng cao (độ phức tạp thuật toán thấp).

Chú ý*:* Trong một số bài toán nếu xây dựng được hàm chọn thích hợp có thể cho nghiệm tối ưu. Trong nhiều bài toán, thuật toán tham ăn chỉ cho nghiệm gần đúng với nghiệm tối ưu.

5.5. Qui hoạch động (Dynamic Programming)

Qui hoạch động là chiến lược xây dựng thuật toán để giải quyết các bài toán tối ưu, có thể đòi hỏi của bài toán không phải là các giá trị quá chi tiết mà chỉ ở dạng giá trị lớn nhất/nhỏ nhất là bao nhiêu chứ không đòi hỏi cụ thể khi nào, ở đâu để có thể đạt được giá trị đó. Trong chiến lược qui hoạch động chúng ta sẽ xây dựng các quan hệ đệ qui của bài toán, bài toán gốc sẽ có lời giải dựa trên các bài toán con (sub problems) dựa trên quan hệ đệ qui. Các thuật toán qui hoạch động thường sử dụng các mảng để lưu lại giá trị nghiệm của các bài toán con và có hai cách tiếp cận: bottom up và top down.

6. Bài tập

Bài tập 1: Xây dựng sơ đồ giải thuật cho bài toán tính số Fibonaci thứ N, biết rằng dãy số Fibonaci được định nghĩa như sau:

F[0] = F[1] = 1, F[N] = F[N-1] + F[N-2] với $N \geq 2$.

Bài tập 2: Xây dựng sơ đồ giải thuật cho bài toán tính biểu thức:

$\sqrt{x+\sqrt{x+...+\sqrt{x}}}$, với N số x thực nằm trong các dấu căn bậc hai, N và x nhập từ bàn phím.

Bài tập 3: Trong một ma trận hai chiều cấp MxN, một phần tử a[i][j] được gọi là điểm yên ngựa của ma trận (saddle point) nếu như nó là phần tử nhỏ nhất trên hàng i và phần tử lớn nhất trên cột j của ma trận. Chẳng hạn a[2][0] = 7 là một phần tử yên ngựa trong ma trận sau:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Hãy viết chương trình tìm tất cả các điểm yên ngựa của một ma trận nhập vào từ bàn phím và đưa ra độ phức tạp của thuật toán.

Bài tập 4: Cho một ma trận kích thước MxN gồm các số nguyên (có cả số âm và dương). Hãy viết chương trình tìm ma trận con của ma trận đã cho sao cho tổng các phần tử trong ma trận con đó lớn nhất có thể được (bài toán maximum sum plateau). Hãy đưa ra đánh giá về độ phức tạp của thuật toán sử dụng.

Bài tập 5: Viết chương trình nhập vào các hệ số của một đa thức (giả sử các hệ số là nguyên và đa thức có biến x là một số nguyên) và một giá trị x0. Hãy tính giá trị của đa thức theo công thức Horner sau:

Nếu $f(x) = a_n*x^n + a_{n-1}*x^{n-1} + .. + a_1*x + a_0$ thì

$f(x) = a_0 + x*(a_1+x*(a_2+x*(….+x(a_{n-1}+a_n*x)…)$ (Công thức Horner).

Bài tập 6: Cho 4 hình hộp kích thước bằng nhau, mỗi mặt của hình hộp được tô bằng 1 trong 4 màu xanh, đỏ, tím, vàng. Hãy đưa ra tất cả các cách xếp các hình hộp thành 1 dãy sao cho khi nhìn theo các phía trên xuống, đằng trước và đằng sau của dãy đều có đủ cả 4 màu xanh, đỏ, tím vàng.

Bài tập 7: Hãy viết chương trình nhanh nhất có thể được để in ra tất cả các số nguyên số có hai chữ số.

Bài tập 8: Áp dụng thuật toán sàng để in ra tất cả các số nguyên tố nhỏ hơn N.

CHƯƠNG II: SẮP XẾP (SORTING) VÀ TÌM KIẾM (SEARCHING)

1. Bài toán sắp xếp

1.1. Sắp xếp trong (Internal Sorting)

Sắp xếp được xem là một trong những lĩnh vực nghiên cứu cổ điển của khoa học máy tính. Trước khi đi vào các thuật toán chi tiết chúng ta cần nắm vững một số khái niệm cơ bản sau:

+ Một trường (field) là một đơn vị dữ liệu nào đó chẳng hạn như tên, tuổi, số điện thoại của một người ...

+ Một bản ghi (record) là một tập hợp các trường

+ Một file là một tập hợp các bản ghi

Sắp xếp (sorting) là một quá trình xếp đặt các bản ghi của một file theo một thứ tự nào đó. Việc xếp đặt này được thực hiện dựa trên một hay nhiều trường nào đó, và các thông tin này được gọi là khóa xắp xếp (key). Thứ tự của các bản ghi được xác định dựa trên các khóa khác nhau và việc sắp xếp đối được thực hiện đối với mỗi khóa theo các thứ tự khác nhau. Chúng ta sẽ tập trung vào các thuật toán xắp xếp và giả sử khóa chỉ gồm 1 trường duy nhất. Hầu hết các thuật toán xắp xếp được gọi là các thuật toán xắp xếp so sánh: chúng sử dụng hai thao tác cơ bản là so sánh và đổi chỗ (swap) các phần tử cần sắp xếp.

Các bài toán sắp xếp đơn giản được chia làm hai dạng.

Sắp xếp trong (internal sorting): Dữ liệu cần sắp xếp được lưu đầy đủ trong bộ nhớ trong để thực hiện thuật toán sắp xếp.

1.2. Sắp xếp ngoài (External Sorting)

Sắp xếp ngoài (external sorting): Dữ liệu cần sắp xếp có kích thước quá lớn và không thể lưu vào bộ nhớ trong để sắp xếp, các thao tác truy cập dữ liệu cũng mất nhiều thời gian hơn.

Trong phạm vi của môn học này chúng ta chỉ xem xét các thuật toán sắp xếp trong. Cụ thể dữ liệu sắp xếp sẽ là một mảng các bản ghi (gồm hai trường chính là trường dữ liệu và trường khóa), và để tập trung vào các thuật toán chúng ta chỉ xem xét các trường khóa của các bản ghi này, các ví dụ minh họa và cài đặt đều được thực hiện trên các mảng số nguyên, coi như là trường khóa của các bản ghi.

1.3. Sắp xếp gián tiếp

Khi các bản ghi có kích thước lớn việc hoán đổi các bản ghi là rất tốn kém, khi đó để giảm chi phí người ta có thể sử dụng các phương pháp sắp xếp gián tiếp. Việc này có thể được thực hiện theo nhiều cách khác nhau và môt trong những phương pháp đó là tạo ra một file mới chứa các trường khóa của file ban đầu, hoặc con trỏ tới hoặc là chỉ số của các bản ghi ban đầu. Chúng ta sẽ sắp xếp trên file mới này với các bản ghi có kích thước nhỏ và sau đó truy cập vào các bản ghi trong file ban đầu thông qua các con trỏ hoặc chỉ số (đây là cách làm thường thấy đối với các hệ quản trị cơ sở dữ liệu).

Ví dụ: chúng ta muốn sắp xếp các bản ghi của file sau đây:

Index	Dept	Last	First	Age	ID number
1	123	Smith	Jon	3	234-45-4586
2	23	Wilson	Pete	4	345-65-0697
3	2	Charles	Philip	9	508-45-6859
4	45	Shilst	Greg	8	234-45-5784

Index	Key
1	123
2	23
3	2
4	45

Sort →

Index	Key
3	2
2	23
4	45
1	123

Sau khi sắp xếp xong để truy cập vào các bản ghi theo thứ tự đã sắp xếp chúng ta sử dụng thứ tự được cung cấp bởi cột index (chỉ số). Trong trường hợp này là 3, 2, 4, 1. (chúng ta không nhất thiết phải hoán đổi các bản ghi ban đầu).

1.3. Các tiêu chuẩn đánh giá một thuật toán sắp xếp

Các thuật toán sắp xếp có thể được so sánh với nhau dựa trên các yếu tố sau đây:

+ Thời gian thực hiện (run-time): số các thao tác thực hiện (thường là số các phép so sánh và hoán đổi các bản ghi).

+ Bộ nhớ sử dụng (Memory): là dung lượng bộ nhớ cần thiết để thực hiện thuật toán ngoài dung lượng bộ nhớ sử dụng để chứa dữ liệu cần sắp xếp.

+ Một vài thuật toán thuộc loại "in place" và không cần (hoặc cần một số cố định) thêm bộ nhớ cho việc thực hiện thuật toán.

+ Các thuật toán khác thường sử dụng thêm bộ nhớ tỉ lệ thuận theo hàm tuyến tính hoặc hàm mũ với kích thước file sắp xếp.

+ Tất nhiên là bộ nhớ sử dụng càng nhỏ càng tốt mặc dù việc cân đối giữa thời gian và bộ nhớ cần thiết có thể là có lợi.

+ Sự ổn định (Stability):Một thuật toán được gọi là ổn định nếu như nó có thể giữ được quan hệ thứ tự của các khóa bằng nhau (không làm thay đổi thứ tự của các khóa bằng nhau).

Chúng ta thường lo lắng nhiều nhất là về thời gian thực hiện của thuật toán vì các thuật toán mà chúng ta bàn về thường sử dụng kích thước bộ nhớ tương đương nhau.

Ví dụ về sắp xếp ổn định: Chúng ta muốn sắp xếp file sau đây dự trên ký tự đầu của các bản ghi và dưới đây là kết quả sắp xếp của các thuật toán ổn định và không ổn định:

File	Stable sorting	Unstable sorting
A Mary	A Mary	A Michel
B Cindy	A Michel	A Peter
A Michel	A Peter	A Mary
B Diana	B Cindy	B Tony
A Peter	B Diana	B Diana
B Tony	B Tony	B Cindy

Chúng ta sẽ xem xét tại sao tính ổn định trong các thuật toán sắp xếp lại được đánh giá quan trọng như vậy.

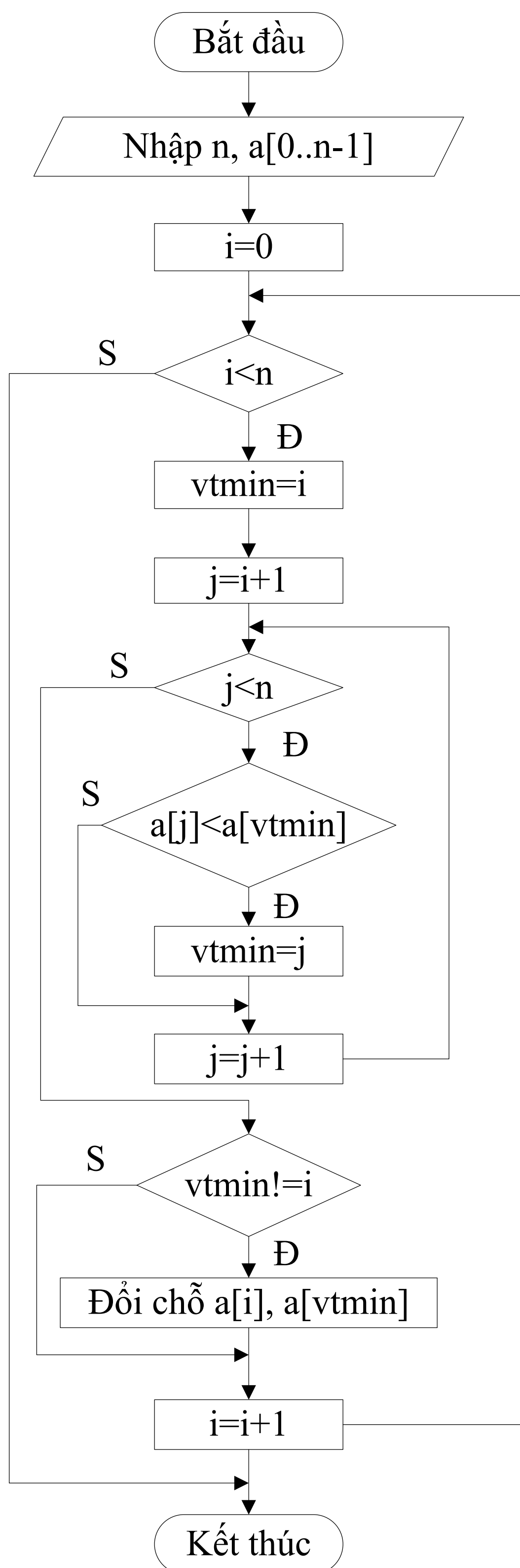
2. Các phương pháp sắp xếp cơ bản

2.1. Sắp xếp chọn (Selection sort)

Mô tả thuật toán:

Tìm phần tử có khóa lớn nhất (nhỏ nhất), đặt nó vào đúng vị trí và sau đó sắp xếp phần còn lại của mảng.

Sơ đồ thuật toán:

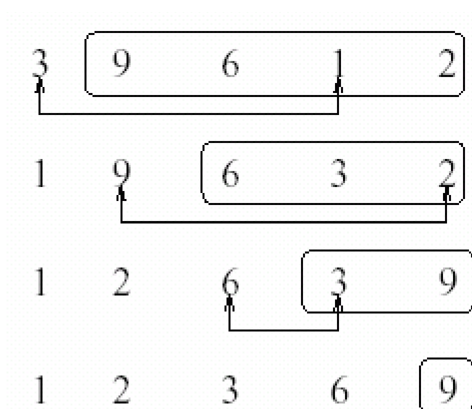


Đoạn mã sau minh họa cho thuật toán:

```
void selection_sort(int a[], int n)
{
  int i, j, vtmin;
```

```
for(i=0; i<n-1;i++)
    {
     vtmin = i; //biến vtmin lưu vị trí phần tử nhỏ nhất a[i..n-1]
     for(j=i+1;j<n;j++)
          if(a[j] < a[vtmin])
               vtmin = j;
     swap(a[vtmin], a[i]); // hàm đổi chỗ a[vtmin], a[i]
    }
}
```

Ví dụ:



Với mỗi giá trị của i thuật toán thực hiện (n – i – 1) phép so sánh và vì i chạy từ 0 cho tới (n–2), thuật toán sẽ cần (n-1) + (n-2) + … + 1 = n(n-1)/2 tức là $O(n^2)$ phép so sánh. Trong mọi trường hợp số lần so sánh của thuật toán là không đổi. Mỗi lần chạy của vòng lặp đối với biến i, có thể có nhiều nhất một lần đổi chỗ hai phần tử nên số lần đổi chỗ nhiều nhất của thuật toán là n. Như vậy trong trường hợp tốt nhất, thuật toán cần 0 lần đổi chỗ, trung bình cần n/2 lần đổi chỗ và tồi nhất cần n lần đổi chỗ.

2.2. Sắp xếp đổi chỗ trực tiếp (Exchange sort)

Tương tự như thuật toán sắp xếp bằng chọn và rất dễ cài đặt (thường bị nhầm với thuật toán sắp xếp chèn) là thuật toán sắp xếp bằng đổi chỗ trực tiếp (một số tài liệu còn gọi là thuật toán Interchange sort hay Straight Selection Sort).

Mô tả: Bắt đầu xét từ phần tử đầu tiên a[i] với i = 0, ta xét tất cả các phần tử đứng sau a[i], gọi là a[j] với j chạy từ i+1 tới n-1 (vị trí cuối cùng). Với mỗi cặp a[i], a[j] đó, để ý là a[j] là phần tử đứng sau a[i], nếu a[j] < a[i], tức là xảy ra sai khác về vị trí thì ta sẽ đổi chỗ a[i], a[j].

Ví dụ minh họa: Giả sử mảng ban đầu là int a[] = {2, 6, 1, 19, 3, 12}. Các bước của thuật toán sẽ được thực hiện như sau:

i=0, j=2: 1, 6, 2, 19, 3, 12

i=1, j=2: 1, 2, 6, 19, 3, 12

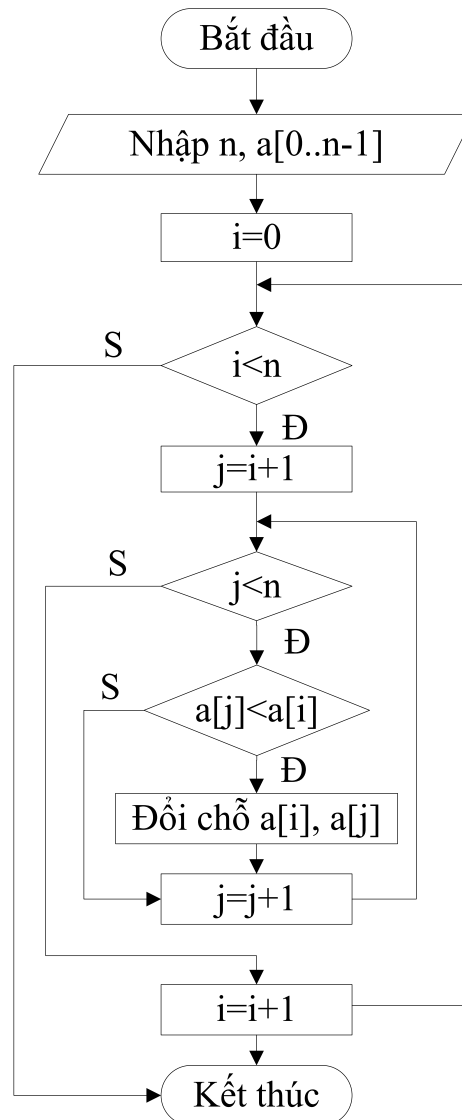
i=2, j=4: 1, 2, 3, 19, 6, 12

i=3, j=4: 1, 2, 3, 6, 19, 12

i=4, j=5: 1, 2, 3, 6, 12, 19

Kết quả cuối cùng: 1, 2, 3, 6, 12, 19.

Sơ đồ thuật toán:



Cài đặt của thuật toán:

```
void exchange_sort(int a[], int n)
{
  int i, j;
        int tam;
  for(i=0; i<n-1;i++)
```

```
            for(j=i+1;j<n;j++)
        if(a[j] < a[i])
        {
                // đổi chỗ a[i], a[j]
                tam = a[i];
                a[i] = a[j];
                a[j] = tam;
            }
}
```

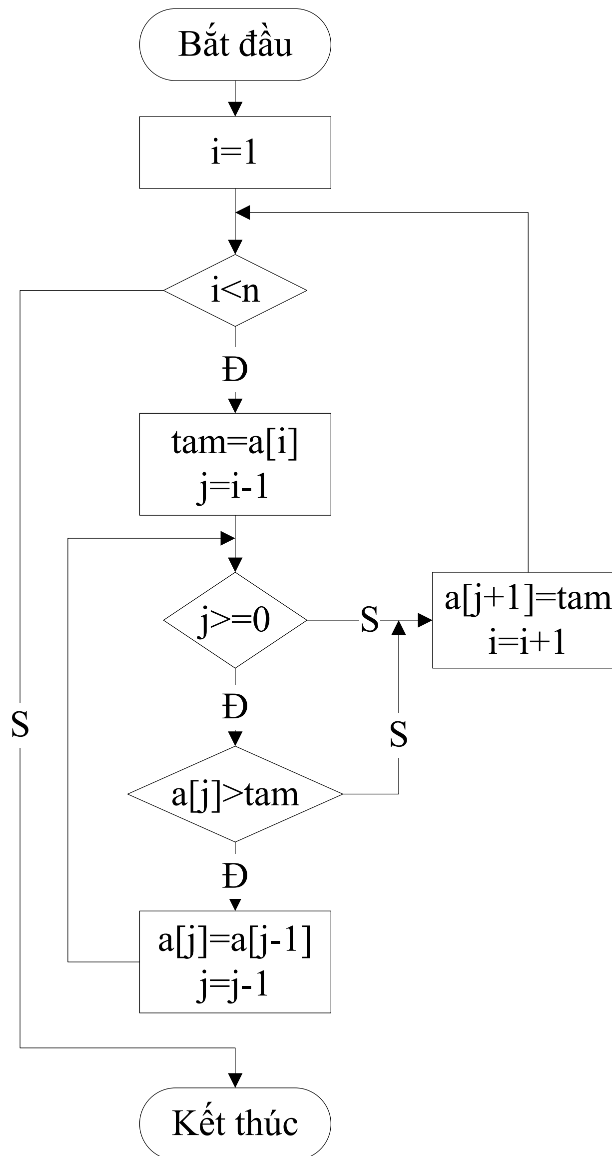
Độ phức tạp của thuật toán: Có thể thấy rằng so với thuật toán sắp xếp chọn, thuật toán sắp xếp bằng đổi chỗ trực tiếp cần số bước so sánh tương đương: tức là n*(n-1)/2 lần so sánh. Nhưng số bước đổi chỗ hai phần tử cũng bằng với số lần so sánh: n*(n-1)/2. Trong trường hợp xấu nhất số bước đổi chỗ của thuật toán bằng với số lần so sánh, trong trường hợp trung bình số bước đổi chỗ là n*(n-1)/4. Còn trong trường hợp tốt nhất, số bước đổi chỗ bằng 0. Như vậy thuật toán sắp xếp đổi chỗ trực tiếp nói chung là chậm hơn nhiều so với thuật toán sắp xếp chọn do số lần đổi chỗ nhiều hơn.

2.3. Sắp xếp chèn (Insertion sort)

Mô tả thuật toán:

Thuật toán dựa vào thao tác chính là chèn mỗi khóa vào một dãy con đã được sắp xếp của dãy cần sắp. Phương pháp này thường được sử dụng trong việc sắp xếp các cây bài trong quá trình chơi bài.

Sơ đồ giải thuật của thuật toán như sau:



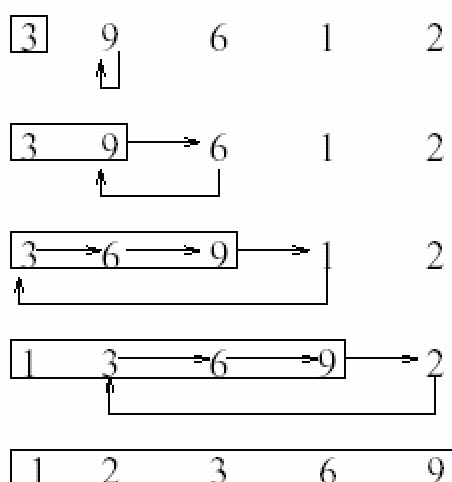
Có thể mô tả thuật toán bằng lời như sau: ban đầu ta coi như mảng a[0..i-1] (gồm i phần tử, trong trường hợp đầu tiên i = 1) là đã được sắp, tại bước thứ i của thuật toán, ta sẽ tiến hành chèn a[i] vào mảng a[0..i-1] sao cho sau khi chèn, các phần tử vẫn tuân theo thứ tự tăng dần. Bước tiếp theo sẽ chèn a[i+1] vào mảng a[0..i] một cách tương tự. Thuật toán cứ thế tiến hành cho tới khi hết mảng (chèn a[n-1] vào mảng a[0..n-2]). Để tiến hành chèn a[i] vào mảng a[0..i-1], ta dùng một biến tạm lưu a[i], sau đó dùng một biến chỉ số j = i-1, dò từ vị trí j cho tới đầu mảng, nếu a[j] > tam thì sẽ copy a[j] vào a[j+1], có nghĩa là lùi mảng lại một vị trí để chèn tam vào mảng. Vòng lặp sẽ kết thúc nếu a[j] < tam hoặc j = -1, khi đó ta gán a[j+1] = tam.

Đoạn mã chương trình như sau:

```
void insertion_sort(int a[], int n)
{
  int i, j, temp;
  for(i=1;i<n;i++)
```

```
    {
        int j = i;
        temp = a[i];
        while(j>0  && temp < a[j-1])
        {
            a[j] = a[j-1];
            j = j-1;
        }
        a[j] = temp;
    }
}
```

Ví dụ:



Thuật toán sắp xếp chèn là một thuật toán sắp xếp ổn định (stable) và là thuật toán nhanh nhất trong số các thuật toán sắp xếp cơ bản.

Với mỗi i chúng ta cần thực hiện so sánh khóa hiện tại (a[i]) với nhiều nhất là i khóa và vì i chạy từ 1 tới n-1 nên chúng ta phải thực hiện nhiều nhất: 1 + 2 + … + n-1 = n(n-1)/2 tức là $\boldsymbol{O}(n^2)$ phép so sánh tương tự như thuật toán sắp xếp chọn. Tuy nhiên vòng lặp while không phải lúc nào cũng được thực hiện và nếu thực hiện thì cũng không nhất định là lặp i lần nên trên thực tế thuật toán sắp xếp chèn nhanh hơn so với thuật toán sắp xếp chọn. Trong trường hợp tốt nhất, thuật toán chỉ cần sử dụng đúng n lần so sánh và 0 lần đổi chỗ. Trên thực tế một mảng bất kỳ gồm nhiều mảng con đã được sắp nên thuật toán chèn hoạt động khá hiệu quả. Thuật toán sắp xếp chèn là thuật toán nhanh nhất trong các thuật toán sắp xếp cơ bản (đều có độ phức tạp $\boldsymbol{O}(n^2)$).

2.4. Sắp xếp nổi bọt (Bubble sort)

Mô tả thuật toán:

Thuật toán sắp xếp nổi bọt dựa trên việc so sánh và đổi chỗ hai phần tử ở kề nhau:

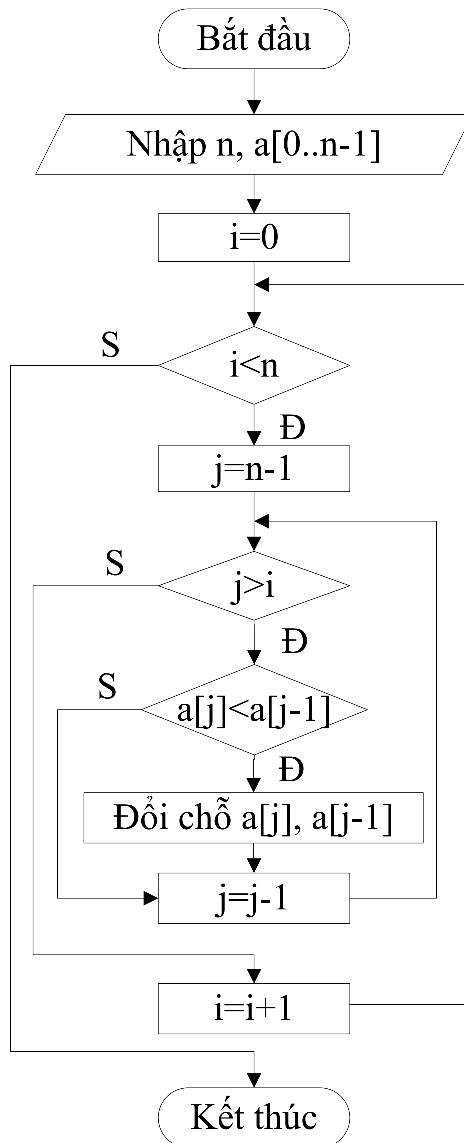
+ Duyệt qua danh sách các bản ghi cần sắp theo thứ tự, đổi chỗ hai phần tử ở kề nhau nếu chúng không theo thứ tự.

+ Lặp lại điều này cho tới khi không có hai bản ghi nào sai thứ tự.

Không khó để thấy rằng n pha thực hiện là đủ cho việc thực hiện xong thuật toán.

Thuật toán này cũng tương tự như thuật toán sắp xếp chọn ngoại trừ việc có thêm nhiều thao tác đổi chỗ hai phần tử.

Sơ đồ thuật toán:



Cài đặt thuật toán:

```
void bubble_sort1(int a[], int n)
{
  int i, j;
```

```
  for(i=n-1;i>=0;i--)
                  for(j=1;j<=i;j++)
                  if(a[j-1]>a[j])
                    swap(a[j-1],a[j]);
}
void bubble_sort2(int a[], int n)
{
  int i, j;
  for(i=0;i<n;i++)
                  for(j=n-1;j>i;j--)
                  if(a[j-1]>a[j])
                    swap(a[j-1],a[j]);
}
```

Thuật toán có độ phức tạp là $\boldsymbol{O}(N*(N-1)/2) = \boldsymbol{O}(N^2)$, bằng số lần so sánh và số lần đổi chỗ nhiều nhất của thuật toán (trong trường hợp tồi nhất). Thuật toán sắp xếp nổi bọt là thuật toán chậm nhất trong số các thuật toán sắp xếp cơ bản, nó còn chậm hơn thuật toán sắp xếp đổi chỗ trực tiếp mặc dù có số lần so sánh bằng nhau, nhưng do đổi chỗ hai phần tử kề nhau nên số lần đổi chỗ nhiều hơn.

2.5. So sánh các thuật toán sắp xếp cơ bản

Sắp xếp chọn:

+ Trung bình đòi hỏi $n^2/2$ phép so sánh, n bước đổi chỗ.

+ Trường hợp xấu nhất tương tự.

Sắp xếp chèn:

+ Trung bình cần n2/4 phép so sánh, n2/8 bước đổi chỗ.

+ Xấu nhất cần gấp đôi các bước so với trường hợp trung bình.

+ Thời gian là tuyến tính đối với các file hầu như đã sắp và là thuật toán nhanh nhất trong số các thuật toán sắp xếp cơ bản.

Sắp xếp nổi bọt:

+ Trung bình cần n2/2 phép so sánh, n2/2 thao tác đổi chỗ.

+ Xấu nhất cũng tương tự.

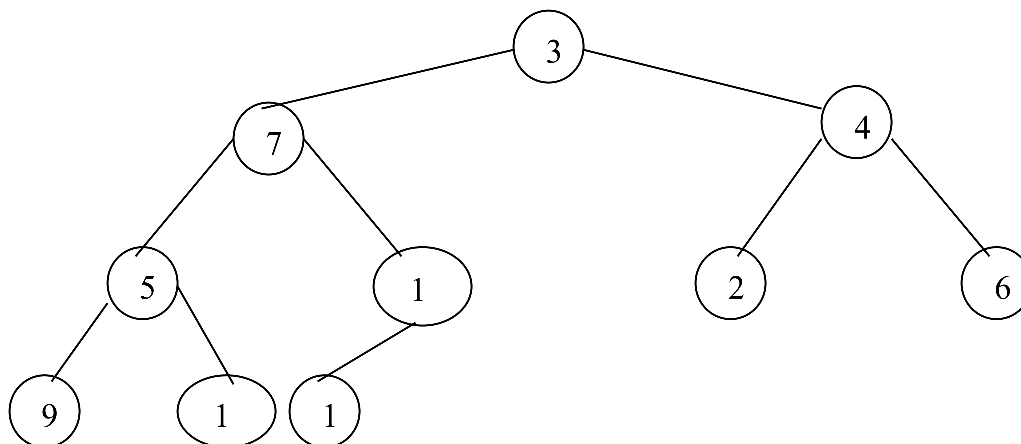
3. Cấu trúc dữ liệu Heap, sắp xếp vun đống (Heap sort).

3.1. Cấu trúc Heap

Trước khi tìm hiểu về thuật toán heap sort chúng ta sẽ tìm hiểu về một cấu trúc đặc biệt gọi là cấu trúc Heap (heap data structure, hay còn gọi là đống).

Heap là một cây nhị phân đầy đủ và tại mỗi nút ta có key(child) ≤ key(parent). Hãy nhớ lại một cây nhị phân đầy đủ là một cây nhị phân đầy ở tất cả các tầng của cây trừ tầng cuối cùng (có thể chỉ đầy về phía trái của cây). Cũng có thể mô tả kỹ hơn là một cây nhị phân mà các nút có đặc điểm sau: nếu đó là một nút trong của cây và không ở mức cuối cùng thì nó sẽ có 2 con, còn nếu đó là một nút ở mức cuối cùng thì nó sẽ không có con nào nếu nút anh em bên trái của nó không có con hoặc chỉ có 1 con và sẽ có thể có con (1 hoặc 2) nếu như nút anh em bên trái của nó có đủ 2 con, nói tóm lại là ở mức cuối cùng một nút nếu có con sẽ có số con ít hơn số con của nút anh em bên trái của nó.

Ví dụ:



Chiều cao của một heap:

Một heap có n nút sẽ có chiều cao là O(log n).

Chứng minh:

Giả sử n là số nút của một heap có chiều cao là h.

Vì một cây nhị phân chiều cho h có số nút tối đa là 2^h -1 nên suy ra:

$2^{h-1} \leq n \leq 2^h - 1$

Lấy logarit hai vế của bất đẳng thức thứ nhất ta được:

$h - 1 \leq \log n$

Thêm 1 vào 2 vế của bất đẳng thức còn lại và lấy logarit hai vế ta lại được:

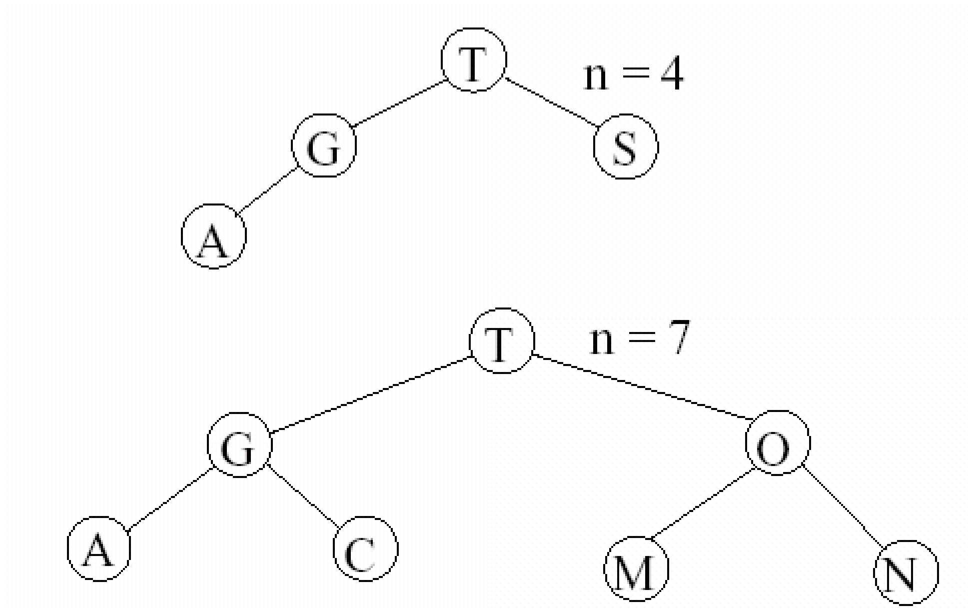
$\log(n + 1) \leq h$

Từ 2 điều trên suy ra:

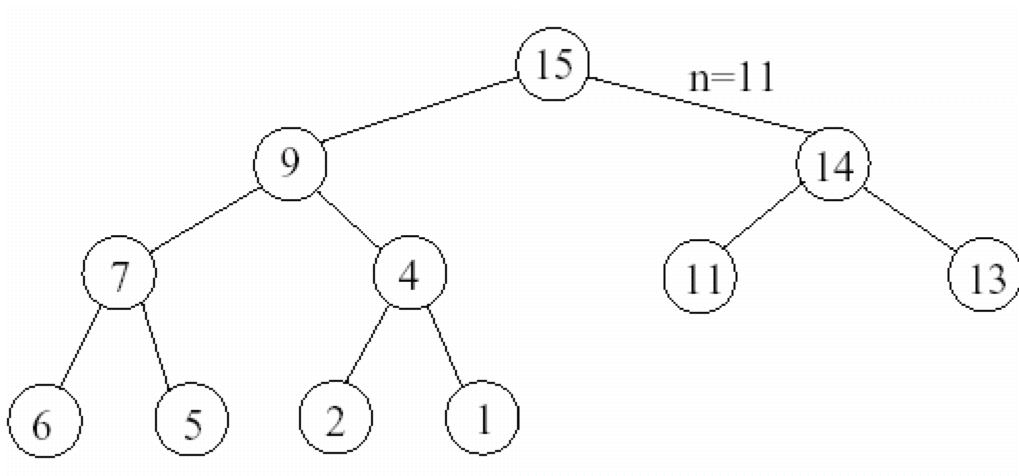
$\log(n+1) \leq h \leq \log(n) + 1$

Các ví dụ về cấu trúc Heap:

Heap với chiều cao h = 3:



heap với chiều cao h = 4



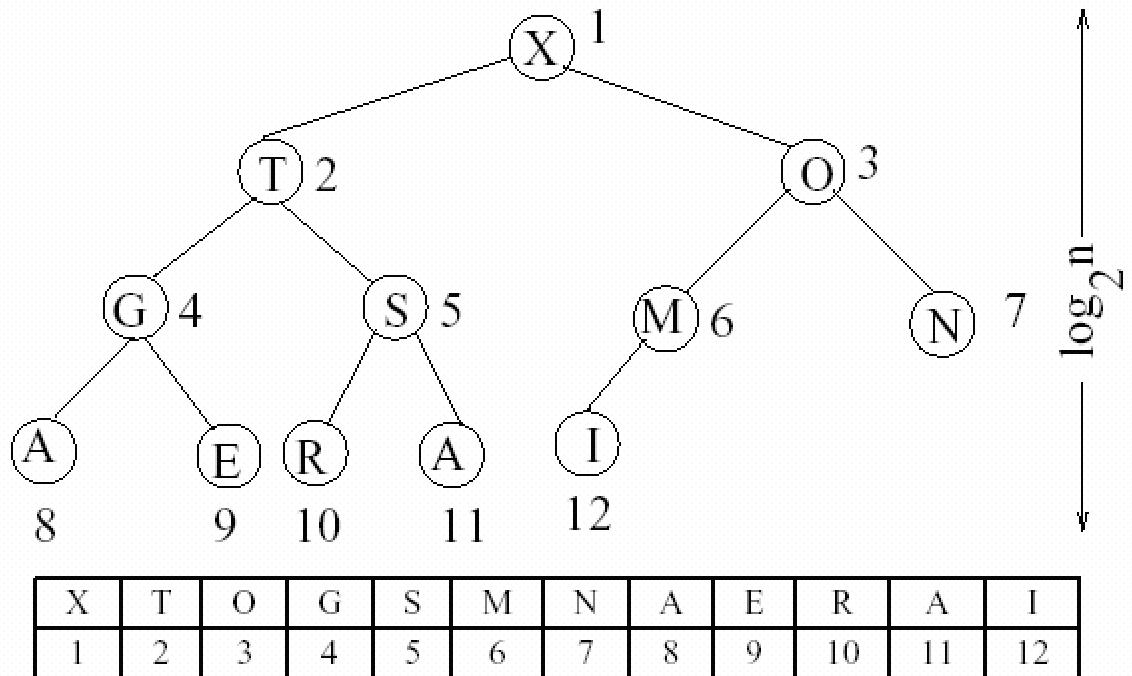
Biểu diễn Heap

Chúng ta đã biết các biểu diễn bằng một cây nhị phân nên việc biểu diễn một heap cũng không quá khó, cũng tương tự giống như biểu diễn một cây nhị phân bằng một mảng.

Đối với một heap lưu trong một mảng chúng ta có quan hệ sau (giả sử chúng ta bắt đầu bằng 0):

- Left(i) = 2*i + 1
- Right(i) = 2*i + 2
- Parent(i) = (i-1)/2

Ví dụ:



X	T	O	G	S	M	N	A	E	R	A	I
1	2	3	4	5	6	7	8	9	10	11	12

Thủ tục heaprify

Đây là thủ tục cơ bản cho tất cả các thủ tục khác thao tác trên các heap

Input:

+ Một mảng A và một chỉ số i trong mảng

+ Giả sử hai cây con Left(i) và Right(i) đều là các heap

+ A[i] có thể phá vỡ cấu trúc Heap khi tạo thành cây với Left(i) và Right(i).

Output:

+ Mảng A trong đó cây có gốc là tại vị trí i là một Heap

Không quá khó để nhận ra rằng thuật toán này có độ phức tạp là ***O***(log n).

Chúng ta sẽ thấy đây là một thủ tục rất hữu ích, tạm thời hãy tưởng tượng là nếu chúng ta thay đổi giá trị của một vài khóa trong heap cấu trúc của heap sẽ bị phá vỡ và điều này đòi hỏi phải có sự sửa đổi.

Sau đây là cài đặt bằng C của thủ tục:

```
void heaprify(int *A, int i, int n)
{
  l = Left(i); /* l = 2*i +1*/
  r = Right(i); /* r = 2*i + 2 */
  if(l < n && A[l] > A[i])
        largest = l;
```

```
    else
            largest = i;
    if(r < n && A[r] > A[largest])
            largest = r;
    if(lagest !=i)
    {
            swap(A[i], A[largest]);
            heaprify(A, largest, n);
    }
}
```

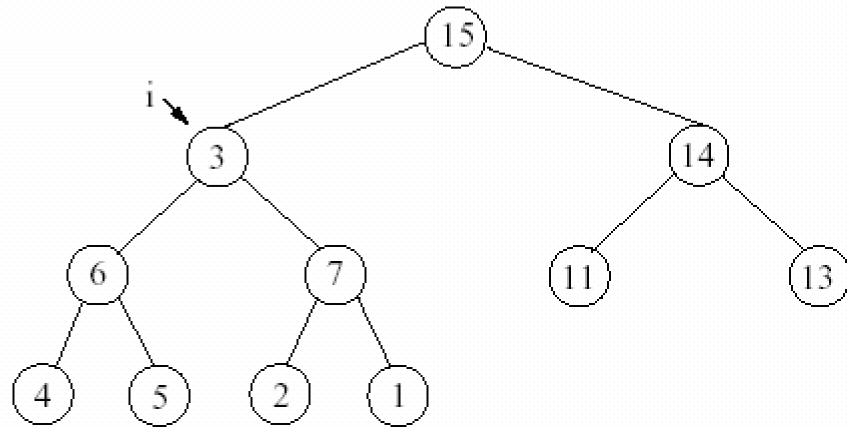
Về cơ bản đây là một thủ tục đơn giản hơn nhiều so với những gì chúng ta cảm nhận về nó. Thủ tục này đơn giản với các bước như sau:

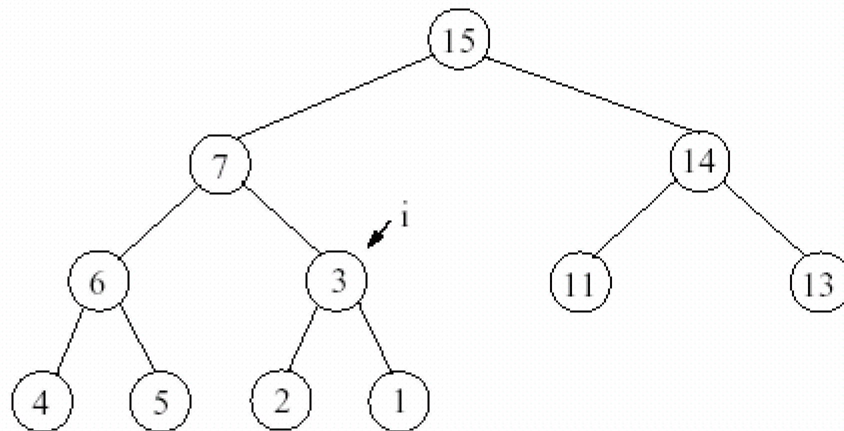
+ Xác định phần tử lớn nhất trong 3 phần tử A[i], A[Left(i)], A[Right(i)].

+ Nếu A[i] không phải là phần tử lớn nhất trong 3 phần tử trên thì đổi chỗ A[i] với A[largest] trong đó A[largest] sẽ là A[Left(i)] hoặc A[Right(i)].

+ Gọi thủ tục với nút largest (vì việc đổi chỗ có thể làm thay đổi tính chất của heap có đỉnh là A[largest]).

Ví dụ:





Thủ tục buildheap

Thủ tục buildheap sẽ chuyển một mảng bất kỳ thành một heap. Về cơ bản thủ tục này thực hiện gọi tới thủ tục heaprify trên các nút theo thứ tự ngược lại. Và vì chạy theo thứ tự ngược lại nên chúng ta biết rằng các cây con có gốc tại các đỉnh con là các heap. Nửa cuối của mảng tương ứng với các nút lá nên chúng ta không cần phải thực hiện thủ tục tạo heap đối với chúng.

Đoạn mã C thực hiện buildheap:

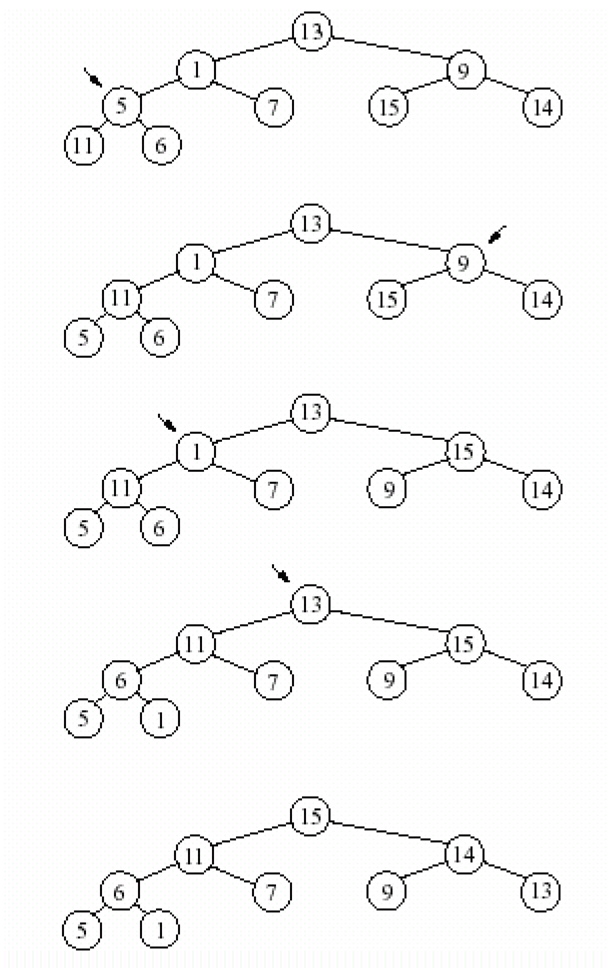
```
void buildheap(int *a, int n)
{
  int i;
  for(i=n/2; i>=0; i--)
          heaprify(a, i, n);
}
```

Dựa và thuật toán này dễ thấy thuật toán tạo heap từ mảng có độ phức tạp là $\boldsymbol{O}$(n*log n). Trên thực tế thuật toán tạo heap có độ phức tạp là $\boldsymbol{O}$(n). Thời gian thực hiện của thuật toán heaprify trên một cây con có kích thước n tại một nút cụ thể i nào đó để chỉnh lại mối quan hệ giữa các phần tử tại a[i], a[Left(i)] và a[Right(i)] là $\boldsymbol{O}$(1). Cộng thêm với thời gian thủ tục này thực hiện trên một cây con có gốc tại một trong các nút là con của nút i. Số cây con của các

con của nút i (i có thể là gốc) nhiều nhất là 2n/3. Suy ra ta có công thức tính độ phức tạp của thuật toán là: T(n) = T(2n/3) + ***O***(1) do đó T(n) = ***O***(log n), từ đây cũng suy ra độ phức tạp của thuật toán buildheap là n*log(n). Cũng có thể lý luận khác như sau: Kích thước của các cấp của cây là: n/4, n/8, n/16, …, 1 trong đó n là số nút của cây. Thời gian để tạo thực hiện thuật toán heaprify đối với các kích thước này nhiều nhất là 1, 2, 3, …, log(n) – 1, vì thế thời gian tổng sẽ xấp xỉ là:

$1*n/4 + 2 * n/8 + 3 * n/16 + \ldots + (\log(n)-1) * 1 < n/4(1 + 2* ½ + 3 * ¼ + 4 * 1/8 + \ldots) = O(n)$.

Ví dụ:



Các thao tác trên heap khác.

Ngoài việc tạo heap các thao tác sau đây cũng thường thực hiện đối với một heap:

+ Insert()

+ Extract_Max()

Chúng ta không bàn về các thao tác này ở đây nhưng các thao tác này đều không khó thực hiện với việc sử dụng thủ tục heaprify mà chúng ta đã cài đặt ở trên. Với các thao tác này chúng ta có thể sử dụng một heap để cài đặt một hàng đợi ưu tiên. Một hàng đợi ưu tiên là

một cấu trúc dữ liệu với các thao tác cơ bản là insert, maximum và extractmaximum và chúng ta sẽ bàn về chúng trong các phần sau của khóa học.

3.2. Sắp xếp vun đống (Heap sort)

Thuật toán Heap sort về ý tưởng rất đơn giản:

+ Thực hiện thủ tục buildheap để biến mảng A thành một heap

+ Vì A là một heap nên phần tử lớn nhất sẽ là A[1].

+ Đổi chỗ A[0] và A[n-1], A[n-1] đã nằm đúng vị trí của nó và vì thế chúng ta có thể bỏ qua nó và coi như mảng bây giờ có kích thước là n-1 và quay trở lại xem xét phần đầu của mảng đã không là một heap nữa.

+ Vì A[0] có thể lỗi vị trí nên ta sẽ gọi thủ tục heaprify đối với nó để chỉnh lại mảng trở thành một heap.

+ Lặp lại các thao tác trên cho tới khi chỉ còn một phần tử trong heap khi đó mảng đã được sắp.

Cài đặt bằng C của thuật toán:

```
void heapsort(int *A, int n)
{
  int i;
  buildheap(A, n);
  for(i=n-1;i>0;i--)
  {
          swap(A[0], A[i]);
          heaprify(A, 0, i-1);
        }
}
```

Chú ý: Để gọi thuật toán sắp xếp trên mảng a có **n** phần tử gọi hàm heapsort() như sau: heapsort(a, n);

Độ phức tạp của thuật toán heapsort:

Thủ tục buildheap có độ phức tạp là O(n).

Thủ tục heaprify có độ phức tạp là O(log n).

Heapsort gọi tới buildheap 1 lần và n-1 lần gọi tới heaprify suy ra độ phức tạp của nó là O(n + (n-1)logn) = O(n*log n).

Trên thực tế heapsort không nhanh hơn quicksort.

4. Tìm kiếm tuyến tính

4.1. Bài toán tìm kiếm

Tìm kiếm là một trong những vấn đề thuộc lĩnh vực nghiên cứu của ngành khoa học máy tính và được ứng dụng rất rộng rãi trên thực tế. Bản thân mỗi con người chúng ta đã có những tri thức, những phương pháp mang tính thực tế, thực hành về vấn đề tìm kiếm. Trong các công việc hàng ngày chúng ta thường xuyên phải tiến hành tìm kiếm: tìm kiếm một cuốn sách để đọc về một vấn đề cần quan tâm, tìm một tài liệu lưu trữ đâu đó trên máy tính hoặc trên mạng, tìm một từ trong từ điển, tìm một bản ghi thỏa mãn các điều kiện nào đó trong một cơ sở dữ liệu, tìm kiếm trên mạng Internet.

Trong môn học này chúng ta quan tâm tới bài toán tìm kiếm trên một mảng, hoặc một danh sách các phần tử cùng kiểu. Thông thường các phần tử này là một bản ghi được phân chia thành hai trường riêng biệt: trường lưu trữ các dữ liệu và một trường để phân biệt các phần tử với nhau (các thông tin trong trường dữ liệu có thể giống nhau hoàn toàn) gọi là trường khóa, tập các phần tử này được gọi là không gian tìm kiếm của bài toán tìm kiếm, không gian tìm kiếm được lưu hoàn toàn trên bộ nhớ của máy tính khi tiến hành tìm kiếm.

Kết quả tìm kiếm là **vị trí của phần tử thỏa mãn điều kiện tìm kiếm**: có trường khóa bằng với một giá trị khóa cho trước (khóa tìm kiếm). Từ vị trí tìm thấy này chúng ta có thể truy cập tới các thông tin khác được chứa trong trường dữ liệu của phần tử tìm thấy. Nếu kết quả là không tìm thấy (trong trường hợp này thuật toán vẫn kết thúc thành công) thì giá trị trả về sẽ được gán cho một giá trị đặc biệt nào đó tương đương với việc không tồn tại phần tử nào có ví trí đó: chẳng hạn như -1 đối với mảng và NULL đối với danh sách liên kết.

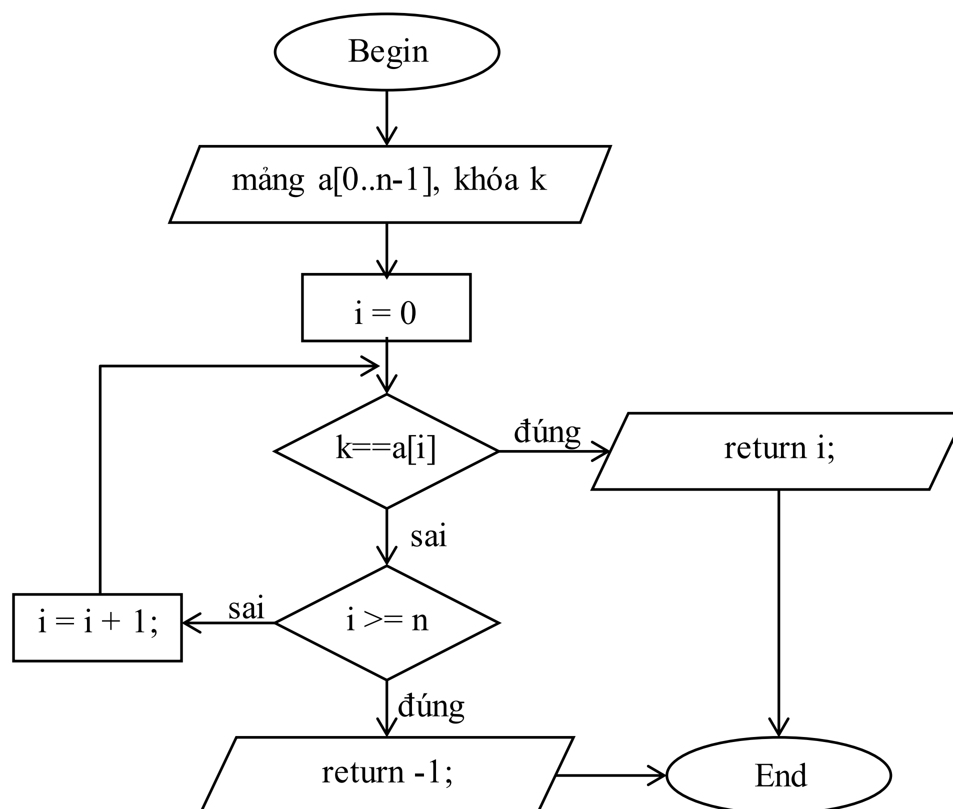
Các thuật toán tìm kiếm cũng có rất nhiều: từ các thuật toán tìm kiếm vét cạn, tìm kiếm tuần tự, tìm kiếm nhị phân, cho tới những thuật toán tìm kiếm dựa trên các cấu trúc dữ liệu đặc biệt như các từ điển, các loại cây như cây tìm kiếm nhị phân, cây cân bằng, cây đỏ đen … Tuy nhiên ở phần này chúng ta sẽ xem xét hai phương pháp tìm kiếm được áp dụng với cấu trúc dữ liệu mảng (dữ liệu tìm kiếm được chứa hoàn toàn trong bộ nhớ của máy tính).

Điều đầu tiên mà chúng ta cần lưu ý là đối với cấu trúc mảng này, việc truy cập tới các phần tử ở các vị trí khác nhau là như nhau và dựa vào chỉ số, tiếp đến chúng ta sẽ tập trung vào thuật toán nên có thể coi như mỗi phần tử chỉ có các trường khóa là các số nguyên.

4.2. Tìm kiếm tuần tự (Sequential search)

Ý tưởng của thuật toán tìm kiếm tuần tự rất đơn giản: duyệt qua tất cả các phần tử của mảng, trong quá trình duyệt nếu tìm thấy phần tử có khóa bằng với khóa tìm kiếm thì trả về vị trí của phần tử đó. Còn nếu duyệt tới hết mảng mà vẫn không có phần tử nào có khóa bằng với khóa tìm kiếm thì trả về -1 (không tìm thấy).

Thuật toán có sơ đồ giải thuật như sau:



Cài đặt bằng C của thuật toán:

```
int sequential_search(int a[], int n, int k)
{
  int i;
  for(i=0;i<n;i++)
          if(a[i]==k)
                  return i;
  return -1;
}
```

Dễ dàng nhận ra thuật toán sẽ trả về kết quả là vị trí của phần tử đầu tiên thỏa mãn điều kiện tìm kiếm nếu tồn tại phần tử đó.

Độ phức tạp thuật toán trong trường hợp trung bình và tồi nhất: $\boldsymbol{O}$(n).

Trong trường hợp tốt nhất thuật toán có độ phức tạp $\boldsymbol{O}$(1).

Các bài toán tìm phần tử lớn nhất và tìm phần tử nhỏ nhất của một mảng, danh sách cũng là thuật toán tìm kiếm tuần tự. Một điều dễ nhận thấy là khi số phần tử của mảng nhỏ (cỡ 10000000) thì thuật toán làm việc ở tốc độ chấp nhận được, nhưng khi số phần tử của mảng lên đến hàng tỷ, chẳng hạn như tìm tên một người trong số tên người của cả thế giới thì thuật toán tỏ ra không hiệu quả.

5. Các vấn đề khác

Ngoài các thuật toán đã được trình bày ở trên vẫn còn có một số thuật toán khác mà chúng ta có thể tham khảo: chẳng hạn như thuật toán sắp xếp Shell sort, sắp xếp bằng đếm Counting sort hoặc sắp xếp cơ số Radix sort. Các thuật toán này được xem như phần tự tìm hiểu của sinh viên.

6. Bài tập

Bài tập 1: Cài đặt các thuật toán sắp xếp cơ bản bằng ngôn ngữ lập trình C trên 1 mảng các số nguyên, dữ liệu của chương trình được nhập vào từ file text được sinh ngẫu nhiên (số phần tử khoảng 10000) và so sánh thời gian thực hiện thực tế của các thuật toán.

Bài tập 2: Cài đặt các thuật toán sắp xếp nâng cao bằng ngôn ngữ C với một mảng các cấu trúc sinh viên (tên: xâu ký tự có độ dài tối đa là 50, tuổi: số nguyên, điểm trung bình: số thức), khóa sắp xếp là trường tên. So sánh thời gian thực hiện của các thuật toán, so sánh với hàm qsort() có sẵn của C.

Bài tập 3[6, trang 52]: Cài đặt của các thuật toán sắp xếp có thể thực hiện theo nhiều cách khác nhau. Hãy viết hàm nhận input là mảng a[0..i] trong đó các phần tử ở chỉ số 0 tới chỉ số i-1 đã được sắp xếp tăng dần, a[i] không chứa phần tử nào, và một số x, chèn x vào mảng a[0..i-1] sao cho sau khi chèn kết quả nhận được là a[0..i] là một mảng được sắp xếp. Sử dụng hàm vừa xây dựng để cài đặt thuật toán sắp xếp chèn.

Gợi ý: Có thể cài đặt thuật toán chèn phần tử vào mảng như phần cài đặt của thuật toán sắp xếp chèn đã được trình bày hoặc sử dụng phương pháp đệ qui.

CHƯƠNG III: ĐỆ QUI VÀ CHIẾN LƯỢC VÉT CẠN

1. Khái niệm đệ qui

Đệ qui là một kỹ thuật được sử dụng trong các ngôn ngữ lập trình cao cấp như C/C++, ngày nay hầu hết các ngôn ngữ lập trình đều hỗ trợ kỹ thuật này. Về bản chất đệ qui là cách định nghĩa một đối tượng dựa trên chính nó, hay cụ thể hơn là trên các thể hiện cụ thể, đơn giản của nó. Ta có thể định nghĩa một bức tranh (picture) như thế này

+ Một điểm ảnh (pixel) là một bức tranh

+ Nếu p1 và p2 là hai bức tranh thì việc ghép p1 với p2 sẽ cho ta một bức tranh mới.

Trong toán học người ta hay sử dụng phương pháp chứng minh qui nạp, chính là một nguyên lý đệ qui. Trong lập trình ta định nghĩa một hàm đệ qui là một hàm mà trong thân hàm (cài đặt) có lời gọi tới chính nó (số lượng và vị trí không hạn chế).

Ví dụ ta có thể định nghĩa hàm tính giai thừa (factorial) của một số nguyên như sau:

Gt(0) = 1.

Gt(n) = n* Gt(n-1) với mọi n > 0.

Giải thuật đệ qui là giải thuật dựa trên các quan hệ đệ qui và được cài đặt cụ thể bằng các hàm đệ qui.

2. Chiến lược vét cạn (Brute force)

Đây là chiến lược đơn giản nhất nhưng cũng là không hiệu quả nhất. Chiến lược vét cạn đơn giản thử tất cả các khả năng xem khả năng nào là nghiệm đúng của bài toán cần giải quyết.

Ví dụ thuật toán duyệt qua mảng để tìm phần tử có giá trị lớn nhất chính là áp dụng chiến lược vét cạn. Hoặc bài toán kiểm tra và in ra tất cả các số nguyên tố có 4 chữ số abcd sao cho ab = cd (các số có 2 chữ số) được thực hiện bằng thuật toán vét cạn như sau:

```
for(a=1;a<=9;a++)
for(b=0;b<=9;b++)
 for(c=0;c<=9;c++)
 for(d=0;d<=9;d++)
      if(ktnguyento(a*1000+b*100+c*10+d) && (10*a+b==10*c+d))
            printf("%d%d%d%d", a, b, c, d);
```

Hàm ktnguyento() kiểm tra xem một số nguyên có phải là số nguyên tố hay không.

Các thuật toán áp dụng chiến lược vét cạn thuộc loại: tìm tất cả các nghiệm có thể có. Về mặt lý thuyết, chiến lược này có thể áp dụng cho mọi loại bài toán, nhưng có một hạn chế khiến nó không phải là chìa khóa vạn năng về mặt thực tế: do cần phải thử tất cả các khả năng

nên số trường hợp cần phải thử của bài toán thường lên tới con số rất lớn và thường quá lâu so với yêu cầu của bài toán đặt ra.

3. Chiến lược quay lui (Back tracking / try and error)

Đây là một trong những chiến lược quan trọng nhất của việc thiết kế thuật toán. Tương tự như chiến lược vét cạn song chiến lược quay lui có một điểm khác: nó lưu giữ các trạng thái trên con đường đi tìm nghiệm của bài toán. Nếu tới một bước nào đó, không thể tiến hành tiếp, thuật toán sẽ thực hiện thao tác quay lui (back tracking) về trạng thái trước đó và lựa chọn các khả năng khác. Bài toán mà loại thuật toán này thường áp dụng là tìm một nghiệm có thể có của bài toán hoặc tìm tất cả các nghiệm sau đó chọn lấy một nghiệm thỏa mãn một điều kiện cụ thể nào đó (chẳng hạn như tối ưu nhất theo một tiêu chí nào đó), hoặc cũng có thể là tìm tất cả các nghiệm của bài toán. Và cũng như chiến lược vét cạn, chiến lược quay lui chỉ có thể áp dụng cho các bài toán kích thước input nhỏ.

Vecto nghiệm

Một trong các dạng bài toán mà chiến lược quay lui thường áp dụng là các bài toán mà nghiệm của chúng là các cấu hình tổ hợp. Tư tưởng chính của giải thuật là xây dựng dần các thành phần của cấu hình bằng cách thử lần lượt tất cả các khả năng có thể có. Nếu tồn tại một khả năng chấp nhận được thì tiến hành bước kế tiếp, trái lại cần lùi lại một bước để thử lại các khả năng chưa được thử. Thông thường giải thuật này thường được gắn liền với cách diễn đạt qui nạp và có thể mô tả chi tiết như sau:

Trước hết ta cần hình thức hóa việc biểu diễn một cấu hình. Thông thường ta có thể trình bày một cấu hình cần xây dựng như là một bộ có thứ tự (vecto) gồm N thành phần:

$X = (x_1, x_2,\ldots,x_N)$

thoả mãn một số điều kiện nào đó.

Giả thiết ta đã xây dựng xong i-1 thành phần x_1, x_2, …, x_{i-1}, bây giờ là bước xây dựng thành phần x_i. Ta lần lượt thử các khả năng có thể có cho x_i. Xảy ra các trường hợp:

Tồn tại một khả năng j chấp nhận được. Khi đó x_i sẽ được xác định theo khả năng này. Nếu x_i là thành phần cuối (i=n) thì đây là một nghiệm, trái lại (i<n) thì tiến hành các bước tiếp theo qui nạp.

Tất cả các khả năng đề cử cho x_i đều không chấp nhận được. Khi đó cần lùi lại bước trước để xác định lại x_{i-1}.

Để đảm bảo cho việc vét cạn (exhausted) tất cả các khả năng có thể có, các giá trị đề cử không được bỏ sót. Mặt khác để đảm bảo việc không trùng lặp, khi quay lui để xác định lại giá trị x_{i-1} cần không được thử lại những giá trị đã thử rồi (cần một kỹ thuật đánh dấu các giá trị đã được thử ở các bước trước).

Trong phần lớn các bài toán, điều kiện chấp nhận j không những chỉ phụ thuộc vào j mà còn phụ thuộc vào việc xác định i-1 thành phần trước, do đó cần tổ chức một số biến trạng thái để cất giữ trạng thái của bài toán sau khi đã xây dựng xong một thành phần đẻ chuẩn bị

cho bước xây dựng tiếp. Trường hợp này cần phải hoàn nguyên lại trạng thái cũ khi quay lui để thử tiếp các khả năng trong bước trước.

Thủ tục đệ qui

Thủ tục cho thuật toán quay lui được thiết kế khá đơn giản theo cơ cấu đệ qui của thủ tục try dưới đây (theo cú pháp ngôn ngữ C).

```
void try(i: integer)
{
        // xác định thành phần xi bằng đệ qui
        int j;
          for j ∈ < tập các khả năng đề cử > do
                if(chấp nhận j)
                {
                        <xác định xi theo khả năng j >
                        < ghi nhận trạng thái mới >
                        If(i=n)
                                < ghi nhận một nghiệm>
                      else
                                try(i+1);
                        < trả lại trạng thái cũ >
                }
}
```

Trong chương trình chính chỉ cần gọi tới try(1) để khởi động cơ cấu đệ qui hoạt động. Tất nhiên, trước đấy cần khởi tạo các giá trị ban đầu cho các biến. Thông thường việc này được thực hiện qua một thủ tục nào đó mà ta gọi là init (khởi tạo).

Hai điểm mấu chốt quyết định độ phức tạp của thuật toán này trong các trường hợp cụ thể là việc xác định các giá trị đề cử tại mỗi bước dành cho xi và xác định điều kiện chấp nhận được cho các giá trị này.

Các giá trị đề cử

Các giá trị đề cử thông thường lớn hơn nhiều so với số các trường hợp có thể chấp nhận được. Sự chênh lệch này càng lớn thì thời gian phải thử càng nhiều, vì thế càng thu hẹp được điều kiện đề cử càng nhiều càng tốt (nhưng không được bỏ sót). Việc này phụ thuộc vào việc phân tích các điều kiện ràng buộc của cấu hình để phát hiện những điều kiện cần của cấu hình đang xây dựng. Lý tưởng nhất là các giá trị đề cử được mặc nhiên chấp nhận. Trong trường hợp này mệnh đề < chấp nhận j > được bỏ qua (vì thế cũng không cần các biến trạng thái).

Ví dụ 1: Sinh các dãy nhị phân độ dài N ($N \leq 20$)

Ví dụ dưới đây trình bày chương trình sinh các dãy nhị phân độ dài N, mỗi dãy nhị phân được tổ chức như một màng n thành phần:

x[0], x[1], …, x[n-1]

trong đó mỗi x[i] có thể lấy một trong các giá trị từ 0 tới 1, có nghĩa là mỗi phần tử x[i] của vecto nghiệm có 2 giá trị đề cử, và vì cần sinh tất cả các xâu nhị phân nên các giá trị đề cử này đều được chấp nhận. Thủ tục chính của chương trình đơn giản như sau:

```
void try(int k)
{
        int j;
  if(k==n)
          in_nghiem();
  else
          for(j=0;j<=1;j++)
          {
                  x[k] = j;
                  try(k+1);
          }
}
```

Trong đó in_nghiem() là hàm in nghiệm tìm được ra màn hình. Dưới đây là toàn bộ chương trình. Trong chương trình có khai báo thêm biến count để đếm các chỉnh hợp được tạo.

CHƯƠNG IV: CHIẾN LƯỢC CHIA ĐỂ TRỊ

1. Cơ sở của chiến lược chia để trị (Divide and Conquer)

Chiến lược chia để trị là một chiến lược quan trọng trong việc thiết kế các giải thuật. Ý tưởng của chiến lược này nghe rất đơn giản và dễ nhận thấy, đó là: khi cần giải quyết một bài toán, ta sẽ tiến hành chia bài toán đó thành các bài toán nhỏ hơn, giải các bài toán nhỏ hơn đó, sau đó kết hợp nghiệm của các bài toán nhỏ hơn đó lại thành nghiệm của bài toán ban đầu.

Tuy nhiên vấn đề khó khăn ở đây nằm ở hai yếu tố: làm thế nào để chia tách bài toán một cách hợp lý thành các bài toán con, vì nếu các bài toán con lại được giải quyết bằng các thuật toán khác nhau thì sẽ rất phức tạp, yếu tố thứ hai là việc kết hợp lời giải của các bài toán con sẽ được thực hiện như thế nào?.

Các thuật toán sắp xếp trộn (merge sort), sắp xếp nhanh (quick sort) đều thuộc loại thuật toán chia để trị (các thuật toán này được trình bày ở chương 3).

Ví dụ: Trong ví dụ này chúng ta sẽ xem xét thuật toán tính a^N.

Để tính a^N ta để ý công thức sau:

f N = 0

$a^N =$ · if N % 2=0

if N % 2 = 1

Từ công thức trên ta suy ra cài đặt của thuật toán như sau:

```
int power(int a, int n)
{
  if(n==0)
        return 1;
  else{
        int t = power(a, n/2);
        if(n%2==0)
              return t*t;
        else
              return a*t*t;
  }
}
```

2. Sắp xếp trộn (Merge sort)

Các phương pháp sắp xếp nâng cao

Các thuật toán sắp xếp tốt nhất đều là các thuật toán đệ qui. Chúng đều tuân theo chiến lược chung sau đây:

Cho một danh sách các bản ghi L.

+ Nếu L có không nhiều hơn 1 phần tử thì có nghĩa là nó đã được sắp

+ Ngược lại

- Chia L thành hai dãy nhỏ hơn là L1, L2
- Sắp xếp L1, L2 (đệ qui – gọi tới thủ tục này)
- Kết hợp L1 và L2 để nhận được L đã sắp

Các thuật toán sắp xếp trộn và sắp xếp nhanh đều sử dụng kỹ thuật này.

Về mặt ý tưởng thuật toán merge sort gồm các bước thực hiện như sau:

+ Chia mảng cần sắp xếp thành 2 nửa

+ Sắp xếp hai nửa đó một cách đệ qui bằng cách gọi tới thủ tục thực hiện chính mergesort

+ Trộn hai nửa đã được sắp để nhận được mảng được sắp.

Đoạn mã C thực hiện thuật toán Merge sort:

```
void mergesort(int *A, int left, int right)
{
  if(left >= right)
        return;
  int mid = (left + right)/2;
  mergesort(A, left, mid);
  mergesort(A, mid+1, right);
  merge(a, left, mid, right);
}
```

Để sắp một mảng a có n phần tử ta gọi hàm như sau: merge_sort(a, 0, n-1);

Nhưng thuật toán trộn làm việc như thế nào?

Ví dụ với 2 mảng sau:

A=	2	4	5	8	10

B=	1	3	6	7	9

C=										

Thuật toán 1:

Thuật toán trộn nhận 2 mảng con đã được sắp và tạo thành một mảng được sắp. Thuật toán 1 làm việc như sau:

+ Đối với mỗi mảng con chúng ta có một con trỏ trỏ tới phần tử đầu tiên

+ Đặt phần tử nhỏ hơn của các phần tử đang xét ở hai mảng vào mảng mới

+ Di chuyển con trỏ của các mảng tới vị trí thích hợp

+ Lặp lại các bước thực hiện trên cho tới khi mảng mới chứa hết các phần tử của hai mảng con

Đoạn mã C++ thực hiện thuật toán trộn hai mảng A, B thành mảng C:

```
int p1 = 0, p2 = 0, index = 0;
int n = sizeA + sizeB;
while(index<n)
{
        if(A[p1] < B[p2]){
                C[index] = A[p1];
                p1++;
                index++;
         }else{
                C[index] = A[p2];
                p2++;
                index++;
        }
}
```

Thuật toán 2:

Thuật toán 1 giả sử chúng ta có 3 mảng phân biệt nhưng trên thực tế chúng ta chỉ có 2 mảng hay chính xác là 2 phần của một mảng lớn sau khi trộn lại thành mảng đã sắp thì đó cũng chính là mảng ban đầu.

Chúng ta sẽ sử dụng thêm một mảng phụ:

```
void merge(int *A, int l, int m, int r)
{
  int *B1 = new int[m-l+1];
  int *B2 = new int[r-m];
  for(int i=0;i<m-l+1;i++)
```

```
            B1[i] = a[i+l];
    for(int i=0;i<r-m;i++)
            B2[i] = a[i+m+l];
    int b1=0,b2=0;
    for(int k=l;k<=r;k++)
            if(B1[b1]<B2[b2])
                    a[k] = B1[b1++];
            else
                    a[k] = B2[b2++];
}
```

Thuật toán 3:

Thuật toán 2 có vẻ khá phù hợp so với mục đích của chúng ta, nhưng cả hai phiên bản của thuật toán trộn trên đều có một nhược điểm chính đó là không kiểm tra các biên của mảng. Có 3 giải pháp chúng ta có thể nghĩ tới:

+ Thực hiện kiểm tra biên một cách cụ thể

+Thêm 1 phần tử lính canh vào đầu của mỗi mảng input.

+ Làm gì đó thông minh hơn

Và đây là một cách để thực hiện điều đó:

```
void merge(int *A,int l,int m,int r)
{
        int *B=new int[r-l+1];
        for (i=m; i>=l; i--)
                B[i-l] = A[i];
        for (j=m+1; j<=r; j++)
                B[j-l] = A[j];
        for (k=l;k<=r;k++)
                if(((B[i] < B[j])&&(i<m))||(j==r+1))
                        A[k]=B[i++];
                else
                        A[k]=B[j--];
}
```

Để sắp xếp mảng a có n phần tử ta gọi hàm như sau:

merge_sort(a, 0, n-1);

Độ phức tạp của thuật toán sắp xếp trộn:

Gọi T(n) là độ phức tạp của thuật toán sắp xếp trộn. Thuật toán luôn chia mảng thành 2 nửa bằng nhau nên độ sâu đệ qui của nó luôn là ***O***(log n). Tại mỗi bước công việc thực hiện có độ phức tạp là ***O***(n) do đó:

T(n) = ***O***(n*log(n)).

Bộ nhớ cần dùng thêm là ***O***(n), đây là một con số chấp nhận được, một trong các đặc điểm nổi bật của thuật toán là tính ổn định của nó, ngoài ra thuật toán này là phù hợp cho các ứng dụng đòi hỏi sắp xếp ngoài.

Chương trình hoàn chỉnh:

```
void merge(int *a,int l,int m,int r)
{
  int *b=new int[r-l+1];
  int i,j,k;
  i = l;
  j = m+1;
  for (k=l;k<=r;k++)
        if((a[i] < a[j])||(j>r))
              b[k]=a[i++];
        else
              b[k]=a[j++];
  for(k=l;k<=r;k++)
        a[k] = b[k];
}
void mergesort(int *a, int l, int r)
{
  int mid;
  if(l>=r)
        return;
  mid = (l+r)/2;
  mergesort(a, l, mid);
  mergesort(a, mid+1, r);
```

```
merge(a, l, mid, r);
}
```

Các chứng minh chặt chẽ về mặt toán học cho kết quả là Merge sort có độ phức tạp là O(n*log(n)). Đây là thuật toán ổn định nhất trong số các thuật toán sắp xếp dựa trên so sánh và đổi chỗ các phần tử, nó cũng rất thích hợp cho việc thiết kế các giải thuật sắp xếp ngoài. So với các thuật toán khác, Merge sort đòi hỏi sử dụng thêm một vùng bộ nhớ bằng với mảng cần sắp xếp.

3. Sắp xếp nhanh (Quick sort)

Quick sort là thuật toán sắp xếp được C. A. R. Hoare đưa ra năm 1962.

Quick sort là một thuật toán sắp xếp dạng chia để trị với các bước thực hiện như sau:

+ Selection: chọn một phần tử gọi là phần tử quay (pivot)

+ Partition (phân hoạch): đặt tất cả các phần tử của mảng nhỏ hơn phần tử quay sang bên trái phần tử quay và tất cả các phần tử lớn hơn phần tử quay sang bên phải phần tử quay. Phần tử quay trở thành phần tử có vị trí đúng trong mảng.

+ Đệ qui: gọi tới chính thủ tục sắp xếp nhanh đối với hai nửa mảng nằm 2 bên phần tử quay

Thuật toán:

```
void quicksort(int *A, int l, int r)
{
  if(r>l)
  {
        int p =partition(A, l, r);
        quicksort(A, l, p -1);
        quicksort(A, p+1, r);
      }
}
```

Hàm phân hoạch partition:

+ Lấy một số k: $l \le k \le r$.

+ Đặt x = A[k] vào vị trí đúng của nó là p

+ Giả sử $A[j] \le A[p]$ nếu $j < p$

+ $A[j] \ge A[p]$ nếu $j > p$

Đây không phải là cách duy nhất để định nghĩa Quicksort. Một vài phiên bản của thuật toán quick sort không sử dụng phần tử quay thay vào đó định nghĩa các mảng con trái và mảng con phải, và giả sử các phần tử của mảng con trái nhỏ hơn các phần tử của mảng con phải.

Chọn lựa phần tử quay

Có rất nhiều cách khác nhau để lựa chọn phần tử quay:

+ Sử dụng phần tử trái nhất để làm phần tử quay

+ Sử dụng phương thức trung bình của 3 để lấy phần tử quay

+ Sử dụng một phần tử ngẫu nhiên làm phần tử quay.

Sau khi chọn phần tử quay làm thế nào để đặt nó vào đúng vị trí và bảo đảm các tính chất của phân hoạch? Có một vài cách để thực hiện điều này và chúng ta sử dụng phương thức chọn phần tử quay là phần tử trái nhất của mảng. Các phương thức khác cũng có thể cài đặt bằng cách sử đổi đôi chút phương thức này.

Hàm phân hoạch:

```
int partition(int *A, int l, int r)
{
  int p = A[l];
  int i = l+1;
  int j = r;
  while(1){
        while(A[i] ≤ p && i<r)
              ++i;
        while(A[j] ≥ p && j>l)
              --j;
        if(i>=j)
        {
              swap(A[j], A[l]);
              return j;
        }else
              swap(A[i], A[j]);
  }
}
```

Để gọi tới hàm trên sắp xếp cho mảng a có n phần tử ta gọi hàm như sau:

quicksort(a, 0, n-1);

Trong thủ tục trên chúng ta chọn phần tử trái nhất của mảng làm phần tử quay, chúng ta duyệt từ hai đầu vào giữa mảng và thực hiện đổi chỗ các phần tử sai vị trí (so với phần tử quay).

Các phương pháp lựa chọn phần tử quay khác:

Phương pháp ngẫu nhiên:

Chúng ta chọn một phần tử ngẫu nhiên làm phần tử quay

Độ phức tạp của thuật toán khi đó không phụ thuộc vào sự phân phối của các phần tử input

Phương pháp 3-trung bình:

Phần tử quay là phần tử được chọn trong số 3 phần tử a[l], a[(l+r)/2] hoặc a[r] gần với trung bình cộng của 3 số nhất.

Hãy suy nghĩ về các vấn đề sau:

Sửa đổi cài đặt của thủ tục phân hoạch lựa chọn phần tử trái nhất để nhận được cài đặt của 2 phương pháp trên

Có cách cài đặt nào tốt hơn không?

Có cách nào tốt hơn để chọn phần tử phân hoạch?

Các vấn đề khác:

Tính đúng đắn của thuật toán, để xem xét tính đúng đắn của thuật toán chúng ta cần xem xét 2 yếu tố: thứ nhất do thuật toán là đệ qui vậy cần xét xem nó có dừng không, thứ hai là khi dừng thì mảng có thực sự đã được sắp hay chưa.

Tính tối ưu của thuật toán. Điều gì sẽ xảy ra nếu như chúng ta sắp xếp các mảng con nhỏ bằng một thuật toán khác? Nếu chúng ta bỏ qua các mảng con nhỏ? Có nghĩa là chúng ta chỉ sử dụng quicksort đối với các mảng con lớn hơn một ngưỡng nào đó và sau đó có thể kết thúc việc sắp xếp bằng một thuật toán khác để tăng tính hiệu quả?

Độ phức tạp của thuật toán:

Thuật toán phân hoạch có thể được thực hiện trong ***O***(n). Chi phí cho các lời gọi tới thủ tục phân hoạch tại bất cứ độ sâu nào theo đệ qui đều có độ phức tạp là ***O***(n). Do đó độ phức tạp của quicksort là độ phức tạp của thời gian phân hoạch độ sau của lời gọi đệ qui xa nhất.

Kết quả chứng minh chặt chẽ về mặt toán học cho thấy Quick sort có độ phức tạp là ***O***(n*log(n)), và trong hầu hết các trường hợp Quick sort là thuật toán sắp xếp nhanh nhất, ngoại trừ trường hợp tồi nhất, khi đó Quick sort còn chậm hơn so với Bubble sort.

4. Tìm kiếm nhị phân

Thuật toán tìm kiếm nhị phân là một thuật toán rất hiệu quả, nhưng điều kiện để áp dụng được thuật toán này là không gian tìm kiếm cần phải được sắp xếp trước theo khóa tìm kiếm.

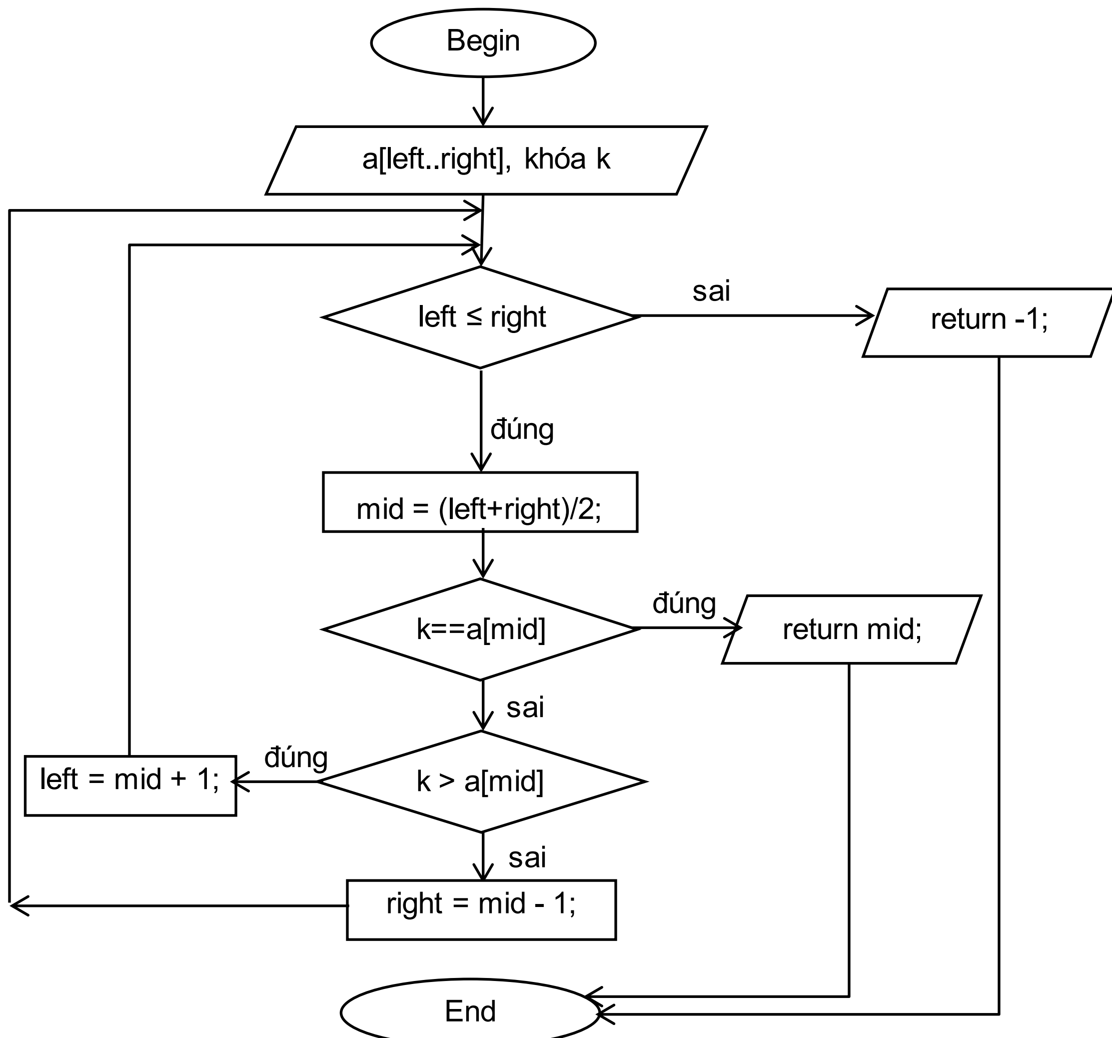
Mô tả thuật toán:

Input: mảng a[left..right] đã được sắp theo khóa tăng dần, khóa tìm kiếm k.

Output: vị trí của phần tử có khóa bằng k.

Thuật toán này thuộc loại thuật toán chia để trị, do mảng đã được sắp xếp, nên tại mỗi bước thay vì duyệt qua các phần tử như thuật toán tìm kiếm tuần tự, thuật toán tìm kiếm nhị phân xét phần tử ở vị trí giữa mảng tìm kiếm a[(left+right)/2], nếu đó là phần tử có khóa bằng với khóa tìm kiếm k thì trả về vị trí đó và kết thúc quá trình tìm kiếm. Nếu không sẽ có hai khả năng xảy ra, một là phần tử đó lớn hơn khóa tìm kiếm k, khi đó do mảng đã đước sắp nên nếu trong mảng có phần tử có trường khóa bằng k thì vị trí của phần tử đó sẽ ở phần trước a[(left+right)/2], có nghĩa là ta sẽ điều chỉnh right = (left+right)/2 - 1. Còn nếu a[(left+right)/2] < k thì theo lý luận tương tự ta sẽ điều chỉnh left = (left+right)/2 + 1. Trong bất cứ trường hợp nào thì không gian tìm kiếm đều sẽ giảm đi một nửa số phần tử sau mỗi bước tìm kiếm.

Sơ đồ thuật toán:



Cài đặt bằng C của thuật toán tìm kiếm nhị phân:

```
int binary_search(int a[], int left, int right, int key)
{
  // cài đặt không đệ qui
  int mid;
  while(left<=right)
  {
        mid = (left + right)/2;
        if(a[mid] == key)
              return mid;
        if(a[mid] < key)
              left = mid + 1;
        else
              right = mid – 1;
  }
  return -1;
}
```

Cài đặt đệ qui:

```
int recursive_bsearch(int a[], int left, int right, int key)
{
  // cài đặt đệ qui
  int mid;
  mid = (left + right)/2;
  if(left>right)
        return -1;
  if(a[mid] == key)
        return mid;
  else
        if(a[mid] < key)
              return recursive_bsearch(a, mid+1, right, key);
        else
```

```
        return recursive_bsearch(a, left, mid-1, key);
}
```

Để gọi hàm cài đặt với mảng a có n phần tử ta gọi như sau:

```
int kq = binary_search(a, 0, n – 1, k);
```

hoặc:

```
int kq = recursive_bsearch(a, 0, n – 1, k);
```

Thuật toán có độ phức tạp là hàm logarit ***O***(log(N)). Với n = 6.000.000.000 thì số thao tác cần thực hiện để tìm ra kết quả là log(n) = 31 thao tác, có nghĩa là chúng ta chỉ cần 31 bước thực hiện để tìm ra tên một người trong số tất cả dân số trên thế giới, thuật toán tìm kiếm nhị phân thực sự là một thuật toán hiệu quả. Trên thực tế việc sắp các từ của từ điển là một áp dụng của thuật toán tìm kiếm nhị phân.

Cách tiếp cận khác để tìm kiếm khi không gian tìm kiếm có số phần tử lớn là xây dựng các cấu trúc cây (chúng ta sẽ xem xét vấn đề này trong chương 5), hoặc sử dụng cấu trúc bảng băm (hash table, không học trong nội dung môn học này) tuy nhiên trong nội dung của môn học này chúng ta chỉ xét hai phương pháp cơ bản trên.

5. Bài tập

Bài tập 1: Viết chương trình nhập vào 1 mảng số nguyên và một số nguyên k, hãy đếm xem có bao nhiêu số bằng k. Nhập tiếp 2 số x < y và đếm xem có bao nhiêu số lớn hơn x và nhỏ hơn y.

Bài tập 2: Cài đặt thuật toán tìm kiếm tuyến tính theo kiểu đệ qui.

Bài tập 3: Viết chương trình nhập một mảng các số nguyên từ bàn phím, nhập 1 số nguyên S, hãy đếm xem có bao nhiêu cặp số của mảng ban đầu có tổng bằng S, có hiệu bằng S.

Bài tập 4: Viết chương trình sinh một dãy các số nguyên. Hãy tìm và đưa ra vị trí, giá trị của số dương đầu tiên trong dãy, số nguyên tố cuối cùng trong dãy.

CHƯƠNG V: QUI HOẠCH ĐỘNG

1. Chiến lược qui hoạch động

Qui hoạch động (DP – Dynamic Programming), một thuật ngữ được nhà toán học Rechard Bellman đưa ra vào năm 1957, là một phương pháp giải bài toán bằng cách kết hợp các lời giải cho các bài toán con của nó giống như phương pháp chia để trị (devide-and-conquer). Các thuật toán chia để trị phân hoạch bài toán cần giải quyết thành các bài toán con độc lập với nhau, sau đó giải quyết chúng bằng phương pháp đệ qui (recursive) và kết hợp các lời giải lại để nhận được lời giải cho bài toán ban đầu. Ngược lại qui hoạch động là phương pháp được áp dụng khi mà các bài toán con của bài toán ban đầu (bài toán gốc) là không độc lập với nhau, chúng có chung các bài toán con nhỏ hơn. Trong các trường hợp như vậy một thuật toán chia để trị sẽ thực hiện nhiều việc hơn những gì thực sự cần thiết, nó sẽ lặp lại việc giải quyết các bài toán con nhỏ hơn đó. Một thuật toán qui hoạch động sẽ chỉ giải quyết tất cả các bài toán con nhỏ một lần duy nhất sau đó lưu kết quả vào một bảng và điều này giúp nó tránh không phải tính toán lại các kết quả mỗi khi gặp một bài toán con nhỏ nào đó.

Qui hoạch động thường được áp dụng với các bài toán tối ưu. Trong các bài toán tối ưu đó thường có nhiều nghiệm (lời giải). Mỗi lời giải có một giá trị được lượng giá bằng cách sử dụng một hàm đánh giá tùy thuộc vào các bài toán cụ thể và yêu cầu của bài toán là tìm ra một nghiệm có giá trị của hàm đánh giá là tối ưu (lớn nhất hoặc nhỏ nhất).

Qui hoạch động là một phương pháp chung rất hiệu quả để giải quyết các vấn đề tối ưu chẳng hạn như trên các đối tượng sắp thứ tự từ trái qua phải, vấn đề tìm đường đi ngắn nhất, vấn đề điều khiển tối ưu … Khi đã hiểu rõ về qui hoạch động việc ứng dụng vào giải các bài toán tối ưu không phải là quá khó khăn nhưng rất nhiều lập trình viên ban đầu phải mất rất nhiều thời gian mới có thể hiểu được. Bài báo này sẽ trình bày các bước cơ bản để áp dụng phương pháp qui hoạch động giải một số các bài toán tối ưu hay được ra trong các kỳ thi Olympic, học sinh giỏi Tin học cấp trường, cấp quốc gia … thông qua từng ví dụ cụ thể.

2. Bài toán 1: Dãy Fibonaci

Bài toán tìm số Fibonaci thứ n là một trong các bài toán quen thuộc đối với những người đã học lập trình, bài toán phát biểu như sau:

Dãy số Fibonaci được cho bởi công thức đệ qui và các điều kiện ban đầu như sau:

$$\begin{cases} F_n = F_{n-1} + F_{n-2}, \forall n \geq 2 \\ F_0 = 0 \\ F_1 = 1 \end{cases}$$

Xây dựng thuật toán tính F_n với n là một số nguyên nhập từ bàn phím.

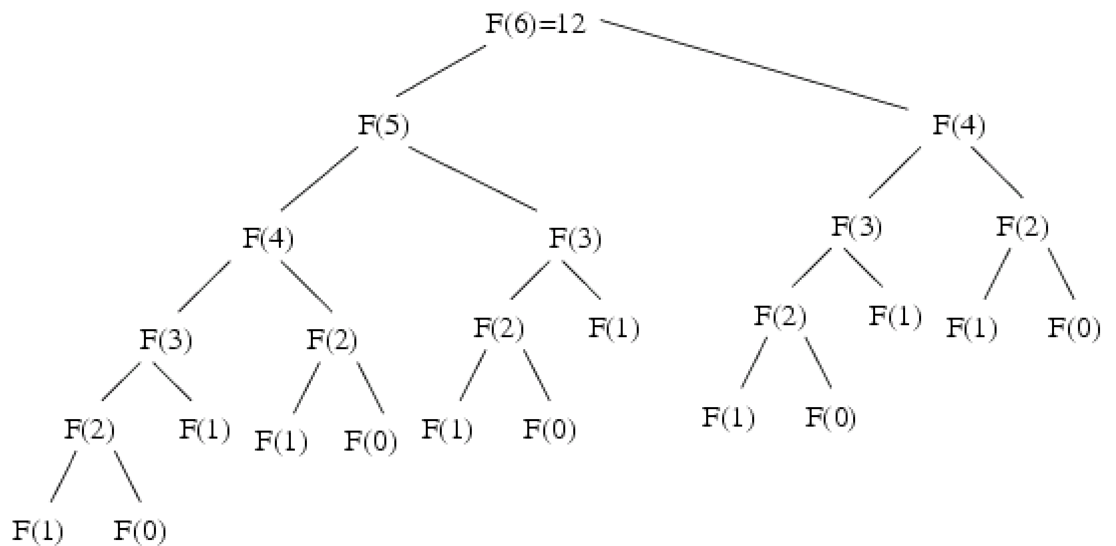
Cách 1: Sử dụng phương pháp đệ qui

Với định nghĩa dãy Fibonaci trên chúng ta dễ dàng thấy thuật toán đơn giản nhất để giải quyết bài toán là sử dụng một hàm đệ qui để tính F_n, chẳng hạn như sau:

int fibonaci(int k)

```
{
  if(k>=2)
          return (fibonaci(k-1)+fibonaci(k-2));
  return 1;
}
```

Thuật toán trên là đúng và hoàn toàn có thể cài đặt với một ngôn ngữ lập trình bất kỳ có hỗ trợ việc sử dụng các hàm đệ qui. Tuy vậy đây là một thuật toán không hiệu quả, giả sử chúng ta cần tính F_6:



Ta có $F_{n+1}/F_n \approx (1+\sqrt{5})/2 \approx 1.61803$ suy ra $F^n \approx 1.61803^n$ và do cây nhị phân tính F_n chỉ có hai nút là F_0 và F_1 nên chúng ta sẽ có xấp xỉ 1.61803^n lần gọi tới hàm fibonaci (trên thực tế n=6 là 13 lần) hay có thể nói độ phức tạp của thuật toán là hàm mũ.

Cách 2: Sử dụng phương pháp qui hoạch động

Chúng ta hoàn toàn có thể sử dụng một thuật toán có độ phức tạp tuyến tính để tính F_n bằng cách sử dụng một mảng để lưu các giá trị dùng để tính F_n:

$F_0 = 0$

$F_1 = 1$

For i = 2 to n

  $F_i = F_{i-1} + F_{i-2}$

Tuy nhiên giảm được thời gian tính toán thì lại mất thêm bộ nhớ để chứa các kết quả trung gian trong quá trình tính toán.

Qua ví dụ 1 chúng ta có một số nhận xét sau:

+ Qui hoạch động là một kỹ thuật tính toán đệ qui hiệu quả bằng cách lưu trữ các kết quả cục bộ.

+ Trong qui hoạch động kết quả của các bài toán con thường được lưu vào một mảng.

3. Bài toán 2: Bài toán nhân dãy các ma trận

Giả sử chúng ta cần nhân một dãy các ma trận: $A \times B \times C \times D \times .....$ Hãy xây dựng thuật toán sao cho số phép nhân cần sử dụng là ít nhất.

Chúng ta biết rằng để nhân một ma trận kích thước $X \times Y$ với một ma trận kích thước $Y \times Z$ (sử dụng thuật toán nhân ma trận bình thường) sẽ cần $X \times Y \times Z$ phép nhân.

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 13 & 18 & 23 \\ 18 & 25 & 32 \\ 23 & 32 & 41 \end{bmatrix}$$

Để giảm số phép nhân cần dùng chúng ta sẽ tránh tạo ra các ma trận trung gian có kích thước lớn và do phép nhân ma trận có tính kết hợp nên có thể đạt được điều này bằng cách sử dụng các dấu đóng mở ngoặc để chỉ ra thứ tự thực phép nhân giữa các ma trận. Bên cạnh đó phép nhân ma trận không phải là đối xứng nên không thể hoán vị thứ tự của chúng mà không làm thay đổi kết quả.

Giả sử chúng ta có 4 ma trận A, B, C, D với các kích thước tương ứng là 30×1, 1×40, 40×10, 10×25. Số giải pháp sử dụng các dấu ngoặc có thể là 3:

((AB)C)D = $30 \times 1 \times 40 + 30 \times 40 \times 10 + 30 \times 10 \times 25 = 20,700$

(AB)(CD) = $30 \times 1 \times 10 + 40 \times 10 \times 25 + 30 \times 40 \times 25 = 41,200$

A((BC)D) = $1 \times 40 \times 10 + 1 \times 10 \times 25 + 30 \times 1 \times 25 = 1400$

Các kết quả trên cho thấy thứ tự thực hiện các ma trận tạo ra một sự sai khác rất lớn về hiệu quả tính toán (số phép nhân cần dùng để tính ra kết quả cuối cùng sai khác nhau rất lớn). Vậy làm thế nào để tìm ra kết quả tối ưu nhất?

Gọi M(i,j) là số lượng phép nhân nhỏ nhất cần thiết để tính $\prod_{k=i}^{j} A_k$ ta có nhận xét sau:

+ Các dấu ngoặc ngoài cùng phân hoạch dãy các ma trận (i,j) tại một vị trí k nào đó.

+ Cả hai nửa của dãy các ma trận (i,j) tại điểm phân hoạch k sẽ đều có thứ tự các dấu ngoặc là tối ưu.

Từ đó ta rút ra công thức sau truy hồi sau:

+ M(i,j) = $Min_{i \le k \le j-1}\left[M(i,k) + M(k+1,j) + d_{i-1}d_k d_j \right]$

+ M(i,i) = 0

Trong đó ma trận A_i có kích thước là (d_{i-1}, d_i).

Cũng giống như trong trường hợp của bài toán Fibonaci nếu chúng ta sử dụng một cài đặt đệ qui thì độ phức tạp của thuật toán sẽ là hàm mũ vì sẽ có rất nhiều lời gọi hàm giống nhau được thực hiện.

Nếu có n ma trận suy ra sẽ có n+1 số nguyên là kích thước của chúng và sẽ có tổ hợp chập 2 của n phần tử các xâu con (mỗi xâu tương ứng với mỗi thứ tự của các dấu ngoặc) giữa 1 và n nên chỉ cần dùng $O(n^2)$ không gian nhớ để lưu kết quả cả các bài toán con.

Lại do k nằm giữa i,j theo phân tích trên nên chúng ta có thể lưu trữ các kết quả trung gian của bài toán trên một ma trận tam giác, đồng thời để chỉ ra nghiệm đạt được kết quả tối ưu cho bài toán chúng ta có thể lưu giá trị k vào một ma trận tam giác cùng kích thước khác.

Thuật toán tính M(1,n):

```
int matrixOrder
{
  for(i=1;i<=n;i++)
          M[i][j] = 0;
  for(b=1;b<n;b++)
          for(i=1;i<=n-b;i++)
          {
                j = i+b;
                M[i][j] = Min_{i≤k≤j-1}[M(i,k) + M(k+1,j) + d_{i-1}d_k d_j]
                faster[i][j] = k;
          }
  return M[1][n];
}
```

Thuật toán in nghiệm:

```
void showOrder(i,j)
{
  if(i==j)
          printf(A[i]);
  else
  {
          k = faster[i][j];
          printf("(");
```

```
            showOrder(i,k);
            printf("*");
            showOrder(k+1,j);
            printf(")");
    }
  }
```

4. Phương pháp qui hoạch động

Qua hai ví dụ trên chúng ta có thể thấy quá trình phát triển của một thuật toán qui hoạch động có thể được chia làm 4 bước như sau:

1. *Xác định đặc điểm cấu trúc của giải pháp tối ưu của bài toán*
2. *Tìm công thức truy hồi (đệ qui) xác định giá trị của một giải pháp tối ưu*
3. *Tính giá trị tối ưu của bài toán dựa vào các giá trị tối ưu của các bài toán con của nó (bottom-up).*
4. *Xây dựng nghiệm đạt giá trị tối ưu từ các thông tin đã tính.*

Các bước 1-3 là các bước cơ bản trong việc giải bất cứ bài toán tối ưu nào bằng phương pháp qui hoạch động. Bước 4 có thể bỏ qua nếu như bài toán chỉ yêu cầu tìm ra giá trị tối ưu chứ không cần chỉ ra nghiệm cụ thể. Thông thường 2 bước đầu là quan trọng và cũng là khó khăn hơn cả, việc xác định cấu trúc nghiệm cũng như công thức truy hồi cần dựa vào kinh nghiệm và sự quan sát các trường hợp cụ thể của bài toán. Do vậy trong quá trình xây dựng thuật toán qui hoạch động cho các bài toán tối ưu chúng ta cần khảo sát các bộ giá trị thực tế của bài toán, giá trị tối ưu và nghiệm của bài toán ứng với các bộ giá trị đó.

Để có thể hiểu rõ hơn quá trình áp dụng các bước của phương pháp qui hoạch động giải bài toán tối ưu chúng ta xét ví dụ thứ 3 sau:

5. Bài toán dãy con chung dài nhất

Cho 2 dãy ký tự X[n] và Y[m] hãy xây dựng thuật toán tìm dãy con chung lớn nhất của hai dãy trên, với dãy con của một dãy được định nghĩa là một tập con các ký tự của dãy (giữ nguyên thứ tự).

Ví dụ với: X = A L G O R I T H M

Y = L O G A R I T H M

Thì dãy con chung dài nhất là Z = LORITHM

Bước 1: Xác định đặc điểm của dãy con chung dài nhất (giải pháp tối ưu của bài toán)

+ Giả sử chúng ta đã có lời giải là Z[1..k]

+ Nếu hai ký tự cuối cùng của X và Y trùng nhau thì đó cũng là ký tự cuối cùng của Z.

+ Phần còn lại của Z khi đó sẽ là xâu con chung dài nhất của X[1..n-1] và Y[1..m-1].

+ Nếu hai ký tự cuối của X và Y không trùng nhau thì một trong số chúng sẽ không nằm trong Z (có thể cả hai).

+ Giả sử ký tự không nằm trong Z trong trường hợp đó là ký tự của X

+ Thế thì Z sẽ là dãy con dài nhất của X[1..n-1] và Y[1..m].

+ Ngược lại nếu ký tự không nằm trong Z là ký tự của Y thì Z sẽ là dãy con dài nhất của X[1..n] và Y[1..m-1].

Bước 2: Xây dựng công thức truy hồi tính độ dài lớn nhất của dãy con của 2 dãy

Từ bước 1 ta có thể tiến hành xây dựng công thức truy hồi như sau:

+ Gọi C[i][j] là độ dài dãy con lớn nhất của hai dãy X[1..i] và Y[1..j]

+ C[i][0] = C[0][j] với mọi i, j.

+ Lời giải của bài toán chính là C[n][m].

+ Công thức truy hồi $C[i][j] = \begin{cases} C[i-1][j-1]+1 (1) \\ \max(C[i-1][j], C[i][j-1]) (2) \end{cases}$

+ Trường hợp 1 là khi X[i] = Y[j], còn trường hợp (2) là khi X[i] khác Y[j]

		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

Bước 3: Xây dựng thuật toán tìm dãy con chung dài nhất của 2 dãy X[1..n] và Y[1..m].

Thứ tự tính toán diễn ra như sau: ban đầu chúng ta tính C[1][1], C[1][2], ..., C[1][n], C[2][1], C[2][2], ..., C[2][n], ... Độ phức tạp để tính C[i][j] bằng $O(1)$ với i=1,n và j=1,m nên độ phức tạp của thuật toán là $O(mn)$.

```
int longest_common_sequence(X, Y)
{
```

```
    for(i=0;i≤m;i++)
        C[i][0] = 0;
    for(j=0;j≤n;j++)
        C[0][j] = 0;
    for(i=1;i≤m;i++)
        for(j=1;j≤n;j++)
            if(X[i]==Y[j])
                C[i][j] = C[i-1][j-1] + 1;
            else
                C[i][j] = max(C[i-1][j],C[i][j-1]);
    return C[m][n];
}
```

		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

Bước 4: Tìm dãy con dài nhất của X[1..n] và Y[1..m](Xây dựng nghiệm đạt giá trị tối ưu từ các thông tin đã tính)

Để tìm lại được nghiệm chúng ta sử dụng một bảng D[i][j] trỏ ngược tới (i, j-1) hoặc (i-1, j) hoặc (i-1, j-1) và lần ngược từ D[m][n] như sau: nếu D[i][j] trỏ tới (i-1, j-1) thì X[i] = Y[j] là ký tự này sẽ nằm trong dãy con dài nhất của 2 xâu. Việc D[i][j] trỏ tới (i-1,j-1), (i-1,j) hoặc (i,j-1) phụ thuộc vào giá trị của mảng C tại vị trí nào được sử dụng để tính C[i][j]. Các giá trị của mảng D sẽ được tính như sau:

D[i][j] bằng 1 (trên trái) nếu C[i][j] = 1 + C[i-1][j-1], bằng 2 (trên) nếu C[i][j] = C[i-1][j] và bằng 3 (trái) nếu C[i][j] = C[i][j-1].

		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

Thuật toán tìm nghiệm: Nếu D[i][j] trỏ tới (i-1, j-1) (1 – trên trái) thì X[i] = Y[j] và ký tự này sẽ nằm trong dãy con là nghiệm.

```
char *  findSolution()
{
        row = m, col = n, lcs=””;
        while((row>0)&&(col>0))
        {
          if(D[row][col]  == 1)
          {
                lcs = lcs + X[row];
                row = row – 1;
                col = col - 1;
          }else{
                      if (D[row][col]==2)
                        row = row – 1;
                else if (D[row][col]  = 3)
                        col = col – 1;
          }
```

```
        }
        reverse lcs; // đảo ngược xâu lcs
        return lcs;
}
```

		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

Qui hoạch động là một phương pháp rất hiệu quả để giải rất nhiều các vấn đề tối ưu hóa thuộc nhiều lĩnh vực khác nhau (Tin học, Kinh tế, Điều khiển, ...) tuy nhiên trong khuôn khổ của bài báo này tác giả chỉ trình bày việc ứng dụng qui hoạch động vào giải một số các bài toán tối ưu mà dạng của chúng thường được ra trong các kỳ thi học sinh giỏi hay Olympic Tin học thông qua 3 ví dụ cụ thể. Hy vọng bài báo giúp được ít nhiều cho độc giả trong việc bước nắm bắt và sử dụng phương pháp qui hoạch động để giải các bài toán tối ưu.

6. Bài tập

Bài 1: ContestSchedule

Vào năm 3006 hầu hết các đều được thực hiện liên quốc gia và hoàn toàn online. Tất cả các cuộc thi lập trình này đều được lên lịch một cách hoàn hảo. Thời gian của mỗi cuộc thi được xác định bởi hai số nguyên s và t tương ứng với thời gian bắt đầu và kết thúc (trước thời điểm t). Vì thế nếu một cuộc thi kết thúc vào thời điểm t=10 và bắt đầu vào thời điểm s=10 thì một lập trình viên có thể tham gia vào cả hai cuộc thi.

Là một lập trình viên có kinh nghiệm nên đối với bất cứ cuộc thi nào Tom cũng có thể dự đoán chính xác tỉ lệ thắng cuộc của mình. Cho trước một danh sách các cuộc thi, hãy giúp Tom tính xem nên tham gia và các cuộc thi nào để tổng tỉ lệ thắng của anh ta là lớn nhất có thể được.

Input

Dữ liệu chương trình được cho trong một file text theo định dạng sau: trên mỗi dòng là dữ liệu về một cuộc thi gồm 3 số nguyên tương ứng là s, t và p (1 ≤ s, t ≤ 1000000, 1 ≤ p ≤ 100) là thời gian bắt đầu, kết thúc và tỉ lệ thắng của cuộc thi.

Output

Kết quả xử lý của chương trình ghi vào 1 file text với độ chính xác đến 5 chữ số sau dấu phẩy.

Ví dụ

Input	Output
1 10 100 **10 20 100** **20 30 100** **30 40 100**	**4.0**
10 20 20 **30 40 60** **15 35 90**	**0.9**
1 100 85 **99 102 100** **101 200 60**	**1.45**

Bài 2: JoinedString

Cho một dãy các từ thành lập từ các chữ cái tiếng Anh, hãy tìm xâu có độ dài nhỏ nhất chứa tất cả các từ trong dãy đã cho. Nếu có nhiều xâu như vậy, hãy đưa ra xâu đầu tiên theo thứ tự Alphabet.

Input

Dữ liệu của chương trình được cho trong một file text, mỗi từ được ghi trên một dòng (số từ nhỏ hơn 13), độ dài của các từ nhỏ hơn 51 và chỉ chứa các ký tự tiếng Anh viết hoa.

Output

Kết quả xử lý của chương trình ghi vào một file text.

Ví dụ

Input	Output
BAB **ABA**	**ABAB**
ABABA **AKAKA** **AKABAS** **ABAKA**	**ABABAKAKABAS**

Bài 3: JumpyNum

Một số Jumpy là một số nguyên dương và các chữ số liền kề của nó khác nhau ít nhất 2 đơn vị. Ví dụ:

Các số thường	28459	28549	1091919	97753
Các số Jumpy	290464	13131313	9753	5

Hãy viết chương trình xác định xem giữa hai số nguyên low, high có bao nhiêu số Jumpy.

Input

Dữ liệu của chương trình được cho trong file text với 2 số low, high (nhỏ hơn 2000000) được ghi trên 2 dòng khác nhau.

Output

Kết quả xử lý của chương trình ghi vào một file text.

Ví dụ

Input	Output
1 **10**	**9**
9 **23**	**9**
8000 **20934**	**3766**

CHƯƠNG VI: CHIẾN LƯỢC THAM LAM (GREEDY)

1. Nguyên tắc tham lam

Các thuật toán tham ăn (greedy algorthims) cũng giống như các thuật toán qui hoạch động thường được sử dụng để giải các bài toán tối ưu (tìm nghiệm của bài toán tốt nhất theo một tiêu chí nào đó). Các thuật toán qui hoạch động luôn cho một nghiệm tối ưu.

Các thuật toán tham ăn thực hiện các lựa chọn tối ưu cục bộ với hy vọng các lựa chọn đó sẽ dẫn đến một nghiệm tối ưu toàn cục cho bài toán cần giải quyết. Các thuật toán tham ăn thường rất dễ cài đặt, nhanh (độ phức tạp thời gian thường là hàm tuyến tính hoặc cùng lắm là bậc 2) , dễ gỡ lỗi và sử dụng ít bộ nhớ. Nhưng thật không may là không phải lúc nào chúng cũng cho các lời giải tối ưu.

Qui hoạch động không hiệu quả

Các thuật toán mà chúng ta đã mới học về qui hoạch động chẳng hạn như thuật toán nhân ma trận ($\boldsymbol{O}(N^2)$), thuật toán tìm xâu con dài nhất ($\boldsymbol{O}(mn)$), thuật toán đối với cây nhị phân tối ưu ($\boldsymbol{O}(N^3)$) đều là các thuật toán xét trên một khía cạnh nào đó vẫn có thể cho là không hiệu quả. Tai sao vậy? Câu trả lời là đối với các thuật toán này chúng ta có rất nhiều lựa chọn trong việc tính toán để tìm ra một giải pháp tối ưu và cần phải thực hiện kiểm tra theo kiểu exshauted đối với tất cả các lựa chọn này. Chúng ta mong muốn có một cách nào đó để quyết định xem lựa chọn nào là tốt nhất hoặc ít nhất cũng hạn chế số lượng các lựa chọn mà chúng ta cần phải thử.

Các thuật toán tham ăn (greedy algorithms) được dùng để giải quyết các bài toán mà chúng ta có thể quyết định đâu là lựa chọn tốt nhất.

Thực hiện lựa chọn theo kiểu tham lam (Greedy Choice).

Mỗi khi cần quyết định xem sẽ chọn lựa chọn nào chúng ta sẽ chọn các lựa chọn tốt nhất vào thời điểm hiện tại (Mỗi khi cần chọn chúng ta sẽ chọn một cách tham lam để maximize lợi nhuận của chúng ta). Tất nhiên các lựa chọn như vậy không phải bao giờ cũng dẫn đến một kết quả đúng. Chẳng hạn cho dãy số <3, 4, 5, 17, 7, 8, 9> cần chọn ra dãy con của nó sao cho dãy con đó là một dãy đơn điệu tăng. Dễ dàng thấy kết quả đúng là <3, 4, 5, 7, 8, 9>. Tuy nhiên theo các chọn tham ăn sau khi chọn xong 3 phần tử đầu là 3, 4, 5 sẽ chọn tiếp phần tử 17, phần tử hợp thành một dãy tăng dài hơn đối với các phần tử đã được chọn trước đó và kết quả sẽ là <3, 4, 5, 17>, tất nhiên kết quả này không phải là kết quả đúng.

2. Bài toán đổi tiền

Bài toán phát biểu như sau: có một lượng tiền n đơn vị (đồng) và k đồng tiền mệnh giá lần lượt là m1, m2, .., mk đơn vị. Với giả thiết số lượng các đồng tiền là không hạn chế, hãy đổi n đồng tiền thành các đồn tiền đã cho với số lượng các đồng tiền sử dụng là ít nhất. Một cách đơn giản theo nguyên lý tham ăn chúng ta thấy rằng để đảm bảo số đồng tiền sử dụng là ít nhất chúng ta sẽ cố gắng sử dụng các đồng tiền có mệnh giá lớn nhiều nhất có thể. Ví dụ với 289 đồng và các đồng tiền mệnh giá 1, 5, 10, 25, 100 chúng ta có thể có một cách đổi là 2 đồng 100, 3 đồng 25, 1 đồng 10 và 4 đồng 1.

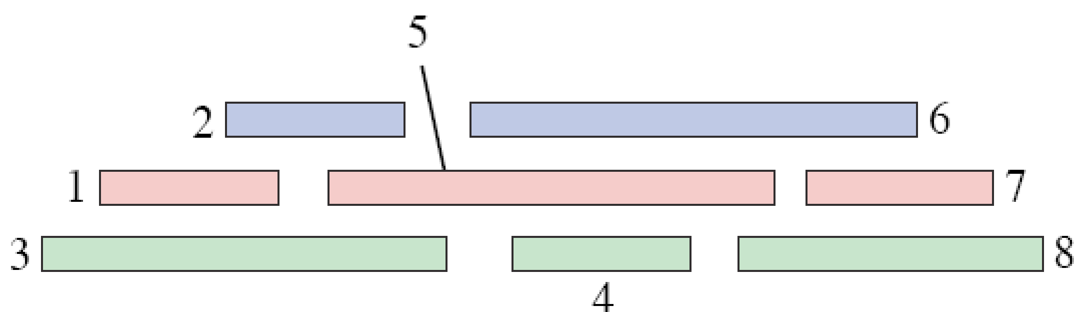
Tuy nhiên trên thực tế bài toán đổi tiền là một bài toán dạng xếp ba lô (knapsack) và thuộc lớp bài toán NP đầy đủ nên thực chất chỉ có 2 cách giải quyết: dùng qui hoạch động với các giá trị nhỏ và duyệt toàn bộ.

3. Bài toán lập lịch

Một phòng học chỉ có thể sử dụng cho một lớp học tại một thời điểm. Có n lớp học muốn sử dụng phòng học, mỗi lớp học có một lịch học được cho bởi một khoảng thời gian $I_j = [s_j, f_j]$ có nghĩa là lớp học sẽ học bắt đầu từ thời điểm s_j tới thời điểm f_j. Mục đích của bài toán là tìm một lịch học sao cho số lượng lớp học có thể sử dụng phòng học là lớn nhất có thể được và tất nhiên không có hai lớp nào cùng sử dụng phòng học tại một thời điểm (không có hai lớp trùng lịch học).

Giả sử lịch học của các lớp được sắp theo thứ tự tăng của thời gian kết thúc như sau:

$f_1 < f_2 < ... < f_n$



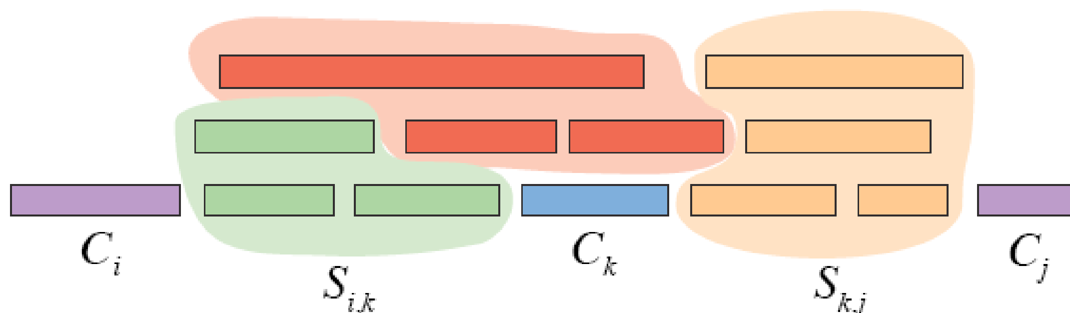
Cấu trúc của một lịch học tối ưu

Gọi $S_{i,j}$ là tập các lớp học bắt đầu sau thời điểm f_i và kết thúc trước thời điểm s_j, có nghĩa là các lớp này có thể được sắp xếp giữa các lớp C_i và C_j.

Chúng ta có thể thêm vào 2 lớp giả định C_0 và C_{n+1} với $f_0 = -\infty$ và $S_{n+1} = +\infty$. Khi đó giá trị $S_{0,n+1}$ sẽ là tập chứa tất cả các lớp.

Giả sử lớp C_k là một phần của lịch học tối ưu của các lớp nằm trong $S_{i,j}$.

Thế thì $i < k < j$ và lịch học tối ưu bao gồm một tập con lớn nhất của $S_{i,k}$, $\{C_k\}$, và một tập con lớn nhất của $S_{k,j}$.



Do đó nếu gọi Q(i, j) là kích thước của một lịch học tối ưu cho tập $S_{i,j}$ chúng ta có công thức sau:

$$Q(i,j) = \begin{cases} 0 & \text{if } j = i+1 \\ \max\limits_{i<k<j}(Q(i,k)+Q(k,j)+1) & \text{if } j > i+1 \end{cases}$$

Thực hiện một lựa chọn tham lam

Bổ đề 1: Tồn tại một lịch học (thời khóa biểu) tối ưu cho tập $S_{i,j}$ chứa lớp C_k trong $S_{i,j}$ kết thúc đầu tiên, có nghĩa là lớp C_k trong $S_{i,j}$ với giá trị chỉ số k nhỏ nhất.

Bổ đề 2: Nếu chọn lớp C_k như bổ đề 1 tập $S_{i,k}$ sẽ là tập rỗng.

Thuật toán tham lam đệ qui

Recursive-Schedule(*S*)

1 **if** |*S*| = 0

2 **then return**

3 Gọi C_k là lớp có thời gian kết thúc nhỏ nhất trong *S*

4 Loại bỏ C_k và tất cả các lớp bắt đầu trước thời gian kết thúc của C_k khỏi *S*;

Gọi *S′* là tập kết quả

5 *O* = Recursive-Schedule(*S′*)

6 **return** $O \cup \{C_k\}$

Dựa trên cấu trúc dữ liệu sử dụng để chứa S, thuật toán sẽ có độ phức tạp thời gian là ***O***(n^2) hoặc ***O***(nlog(n)). Thêm vào đó thuật toán sẽ mất một chi phí cho việc bảo trì ngăn xếp vì nó là đệ qui.

Thuật toán tham ăn thời gian tuyến tính theo kiểu lặp

Iterative-Schedule(*S*)

1 $n = |S|$

2 $m = -\infty$

3 $O = \{\}$

3 **for** $i = 1..n$

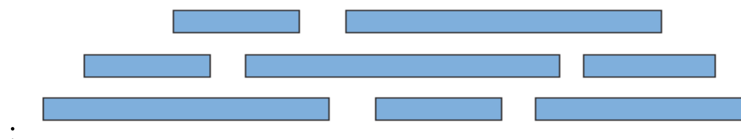
4 **do if** $s_i \geq m$

5 **then** $O = O \cup \{C_i\}$

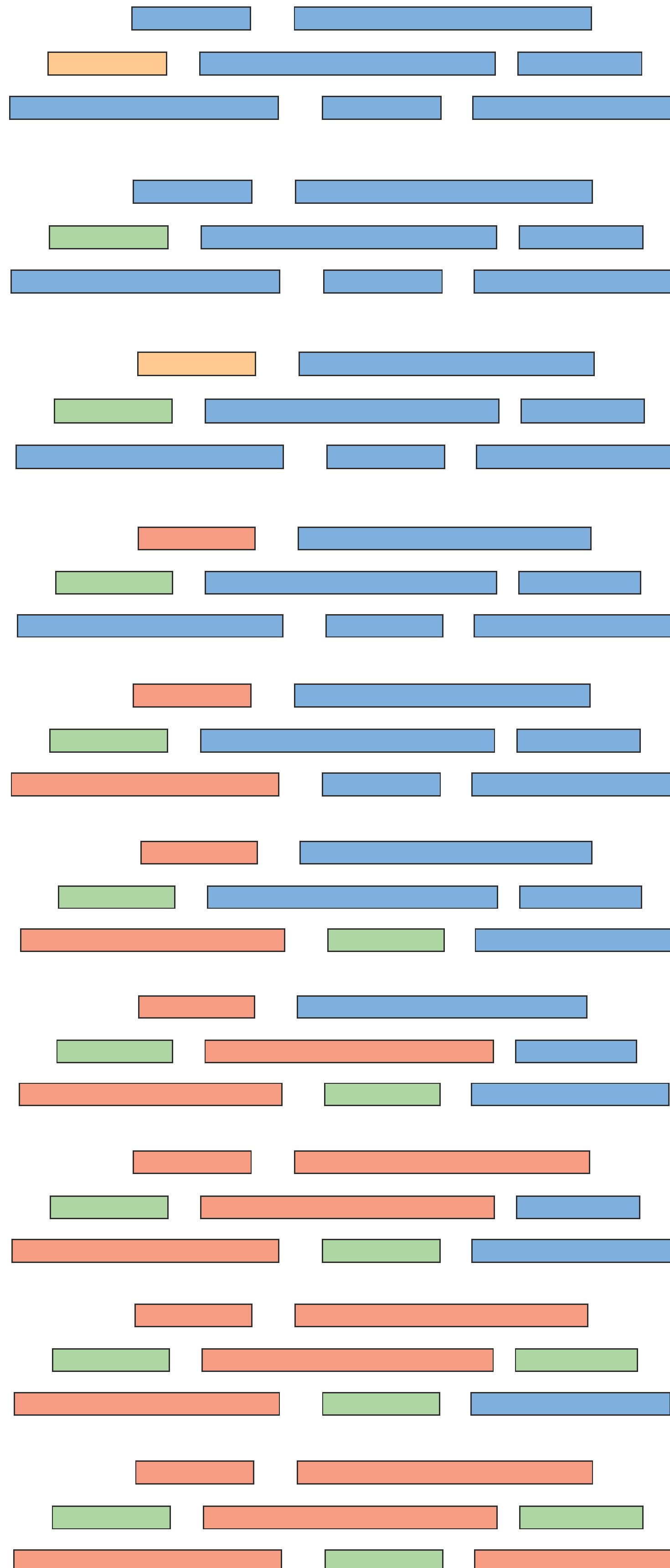
6 $m = f_i$

7 **return** *O*

Minh họa quá trình thực hiện của thuật toán



:



Thuật toán trên hiển nhiên có thời gian thực hiện tuyến tính và nó là đúng đắn dựa trên bổ đề sau: "Gọi O là tập các lớp hiện tại, và C_k là lớp cuối cùng được thêm vào O. Thế thì đối với lớp C_l bất kỳ mà $l > k$, nếu C_l xung đột với một lớp trong O nó sẽ xung đột với C_k.

Các biến thể của bài toán: Thay vì tìm số lượng lớn nhất các lớp có thể sử dụng phòng học chúng ta có thể thay đổi yêu cầu của bài toán là tìm thời gian lớn nhất mà phòng học được sử dụng. Cách tiếp cận qui hoạch động khi chúng ta thay đổi mục tiêu của bài toán là không đổi nhưng các lựa chọn tham ăn của theo kiểu như trên sẽ không làm việc được đối với bài toán này.

4. So sánh chiến lược tham lam và qui hoạch động

Vậy khi nào thì thuật toán tham ăn sẽ cho nghiệm tối ưu?

Bài toán được giải quyết bằng thuật toán tham ăn (thường là các bài toán tối ưu) nếu như nó có hai đặc điểm sau:

+ Tính lựa chọn tham ăn (greedy choice property): Một nghiệm tối ưu có thể nhận được bằng cách thực hiện các lựa chọn có vẻ như là tốt nhất tại mỗi thời điểm mà không cần quan tâm tới các gợi ý của nó đối với các nghiệm của các bài toán con. Hay nói một cách dễ hiểu là một nghiệm tối ưu của bài toán có thể nhận được bằng cách thực hiện các lựa chọn tối ưu cục bộ.

+ Một nghiệm tối ưu có thể nhận được bằng cách gia tăng (augmenting) các nghiệm thành phần đã được xây dựng với một nghiệm tối ưu của bài toán con còn lại. Có nghĩa là một nghiệm tối ưu sẽ chứa các nghiệm tối ưu đối với các bài toán con kích thước nhỏ hơn. Tính chất này được gọi là cấu trúc con tối ưu (optimal substructure).

Sự khác nhau cơ bản giữa các thuật toán qui hoạch động và các thuật toán tham ăn đó là đối với các thuật toán tham ăn chúng ta không cần biết các nghiệm của các bài toán con để có thể thực hiện một lựa chọn nào đó. Và vì một bài toán có thể có rất nhiều các nghiệm tối ưu khác nhau nên các thuật toán tham ăn có thể hiệu quả hơn các thuật toán qui hoạch động.

Chú ý: *Trong một số bài toán nếu xây dựng được hàm chọn thích hợp có thể cho nghiệm tối ưu. Trong nhiều bài toán, thuật toán tham ăn chỉ cho nghiệm gần đúng với nghiệm tối ưu.*

TÀI LIỆU THAM KHẢO

1. Wikipedia, “Từ điển bách khoa toàn thư trực tuyến tiếng Việt”, http://vi.wikipedia.org/wiki/.
2. Wikipedia, “Từ điển bách khoa toàn thư trực tuyến tiếng Anh”, http://en.wikipedia.org/wiki/Main_Page.
3. Các tài liệu và bài giảng tại website: http://csce.unl.edu/~cusack/Teaching/?page=notes.
4. Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest and Clifford Stein, “Introduction to Algorithms, Second Edition”, The MIT Press, 2001, 1180 pages.
5. Jeff Cogswell, Christopher Diggins, Ryan Stephens, Jonathan Turkanis, “C++ Cookbook”, O’Reilly, November 2005, 592 pages.
6. Nguyễn Hữu Điển, Giáo trình một số vấn đề về thuật toán, NXB Giáo dục, 2003
7. Đinh Mạnh Tường. Cấu trúc dữ liệu và thuật toán. NXB Đại học Quốc gia Hà nội. 2002.

ĐỀ THI THAM KHẢO

Đề số 1:

Câu 1:

a) Thế nào là bài toán tìm kiếm? Hãy trình bày các bước, đánh giá độ phức tạp và vẽ sơ đồ của thuật toán tìm kiếm tuyến tính?

b) Viết hàm cài đặt thuật toán sắp xếp nổi bọt tăng dần trên mảng cấu trúc công nhân gồm các trường thông tin sau:

- Tên
- Tuổi
- Lương tháng

Trường khóa để sắp xếp là trường lương, nếu cùng lương thì theo tên.

Câu 2:

a) Trình bày và cài đặt thuật toán sinh xâu số từ các số 1, 2, 3 với độ dài n nhập từ bàn phím.

b) Thực hiện các bước của thuật toán sắp xếp trộn với mảng số nguyên sau: 3, 8, 10, 9, 82, 4, 78, 28, 9, 10, 13, 11.

Câu 3:

a) Trình bày thuật toán nhân dãy ma trận và áp dụng tìm số phép nhân ít nhất để thực hiện nhân dãy các ma trận có kích thước: 5x50, 50x10, 10x20, 20x15

b) Trình bày thuật toán tìm dãy con gồm các phần tử liên tiếp có tổng lớn nhất của dãy số nguyên sau: -9, 8, -3, 18, 4, -2, 8, -13, 20, -4, 8, 9, 3.

Đề số 2:

Câu 1:

a) Thế nào là bài toán tìm kiếm? Hãy trình bày các bước, đánh giá độ phức tạp và vẽ sơ đồ của thuật toán tìm kiếm nhị phân?

b) Viết hàm cài đặt thuật toán sắp xếp chọn tăng dần trên mảng cấu trúc công nhân gồm các trường thông tin sau:

- Tên
- Hệ số lương
- Phụ cấp

Trường khóa để sắp xếp là lương = hệ số lương * 750 + phụ cấp.

Câu 2:

a) Trình bày và cài đặt thuật toán sinh xâu số từ các số 1, 2, 3, 6 với độ dài n nhập từ bàn phím.

b) Thực hiện các bước của thuật toán sắp xếp nhanh với mảng số nguyên sau: 3, 8, 10, 9, 82, 4, 78, 28, 9, 10, 13, 11.

Câu 3:

a) Trình bày thuật toán nhân dãy ma trận và áp dụng tìm số phép nhân ít nhất để thực hiện nhân dãy các ma trận có kích thước: 15x5, 5x20, 20x10, 10x7.
b) Trình bày thuật toán tìm dãy con gồm các phần tử liên tiếp có tổng lớn nhất của dãy số nguyên sau: -8, 9, 7, -2, -19, 2, -9, 2, 3, 28, -9.

Đề số 3:

Câu 1:

a) Thế nào là bài toán tìm kiếm? Hãy trình bày các bước, đánh giá độ phức tạp và vẽ sơ đồ của thuật toán tìm kiếm tuyến tính?
b) Viết hàm cài đặt thuật toán sắp xếp chèn tăng dần trên mảng cấu trúc công nhân gồm các trường thông tin sau:

- Tên
- Tuổi
- Lương tháng

Trường khóa để sắp xếp là trường lương, nếu cùng lương thì theo tuổi.

Câu 2:

a) Trình bày và cài đặt thuật toán sinh các hoán vị của số nguyên đầu tiên với n nhập từ bàn phím.
b) Thực hiện các bước của thuật toán sắp xếp trộn với mảng số nguyên sau: 3, 8, 10, 9, 82, 4, 78, 28, 9, 10, 13, 11.

Câu 3:

a) Trình bày thuật toán nhân dãy ma trận và áp dụng tìm số phép nhân ít nhất để thực hiện nhân dãy các ma trận có kích thước: 5x50, 50x10, 10x20, 20x15

Đề số 4:

Câu 1:

a) Thế nào là bài toán tìm kiếm? Hãy trình bày các bước, đánh giá độ phức tạp và vẽ sơ đồ của thuật toán tìm kiếm nhị phân?
b) Viết hàm cài đặt thuật toán sắp xếp đổi chỗ trực tiếp tăng dần trên mảng cấu trúc công nhân gồm các trường thông tin sau:

- Tên
- Tuổi
- Lương tháng

Trường khóa để sắp xếp là trường lương, nếu cùng lương thì theo tuổi.

Câu 2:

a) Trình bày và cài đặt thuật toán sinh xâu số từ các số 2, 3, 5 với độ dài n nhập từ bàn phím.
b) Thực hiện các bước của thuật toán sắp xếp nhanh với mảng số nguyên sau: 3, 8, 10, 9, 82, 4, 78, 28, 9, 10, 13, 11.

Câu 3:

a) Trình bày thuật toán nhân dãy ma trận và áp dụng tìm số phép nhân ít nhất để thực hiện nhân dãy các ma trận có kích thước: 5x50, 50x10, 10x20, 20x15.
b) Áp dụng thuật toán tìm xâu con chung dài nhất của hai xâu X1 = "ABCABCCB" và X2="BCAABCCA".

Đề số 5:

Câu 1:

a) Thế nào là bài toán tìm kiếm? Hãy trình bày các bước, đánh giá độ phức tạp và vẽ sơ đồ của thuật toán tìm kiếm tuyến tính?
b) Viết hàm cài đặt thuật toán sắp xếp nổi bọt tăng dần trên mảng cấu trúc Học sinh gồm các trường thông tin sau:

- Tên
- Tuổi
- Điểm trung bình

Trường khóa để sắp xếp là trường điểm trung bình, nếu cùng điểm trung bình thì theo tuổi.

Câu 2:

a) Trình bày và cài đặt thuật toán sinh xâu số từ các số 3, 7, 2 với độ dài n nhập từ bàn phím.
b) Thực hiện các bước của thuật toán sắp xếp trộn với mảng số nguyên sau: 3, 8, 10, 9, 82, 4, 78, 28, 9, 10, 13, 11.

Câu 3:

a) Trình bày thuật toán nhân dãy ma trận và áp dụng tìm số phép nhân ít nhất để thực hiện nhân dãy các ma trận có kích thước: 15x5, 5x20, 20x10, 10x15
b) Trình bày thuật toán tìm xâu con chung dài nhất của hai xâu con.